

Конспект онлайн-курса «Операционные системы семейства UNIX. Системное программирование» от НИУ ВШЭ

Предисловие

Конспект онлайн-курса [«Операционные системы семейства UNIX. Системное программирование»](#) от НИУ ВШЭ основан на следующих материалах:

- Лекции и презентации курса.
- [Конспект курса «Операционные системы» студентов 3-го курса МИЭМ \(Московский государственный институт электроники и математики\)](#).
- [Истратов А.Ю. Межпроцессное взаимодействие на уровне «клиент-сервер» в ОС UNIX \(Учебное пособие\)](#). — М.: МИЭМ, 2009 г.
- [Истратов А.Ю. Программирование на командном языке в операционной среде UNIX](#). — М.: МИЭМ, 2014 г.
- Стивен Р.У., Раго С.А. UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018 г.
- Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX: Руководство программиста по разработке ПО — М.: ДМК Пресс, 2000 г.
- Робачевский А., Немнюгин С., Стесик О. Операционная система UNIX. 2-е изд. — СПб.: БХВ-Петербург, 2010 г.
- Керриск М. Linux API. Исчерпывающее руководство. — СПб.: Питер, 2019 г.
- Лав Р. Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014 г.
- [Истратов А.Ю. Программирование в операционной среде UNIX: обмен информацией между параллельными процессами, организация защиты файлов в файловой системе, обработка прерываний](#). — М.: МИЭМ, 2009 г.
- [Родионов С.В., Князев В.Д. Базовые концепции OS UNIX: Методическое руководство по учебному курсу «Операционные системы ЭВМ»](#). — М.: МГТУ им. Н.Э. Баумана.
- [М.А. Казачук, И.В. Машечкин, И.С. Попов, А.Н. Терехин, В.В Тюляева. Программирование в ОС UNIX на языке Си: учебно-методическое пособие. 2-е изд. испр. и доп.](#) — М.: МГУ 2020 г.

Составителем конспекта исправлены все ошибки и неточности, обнаруженные в лекциях и презентациях курса, включая примеры исходного кода и иллюстрации.

В тексте конспекта курсивом выделена информация, выходящая за рамки онлайн-курса, необязательная при подготовке к экзамену. Кроме того, отдельно представлен раздел «Не вошедшее (материалы очного курса)», материалы которого также выходят за рамки онлайн-курса (они позаимствованы из [конспекта студентов 3-го курса МИЭМ](#)).

1. Понятие «Операционная система» (ОС)

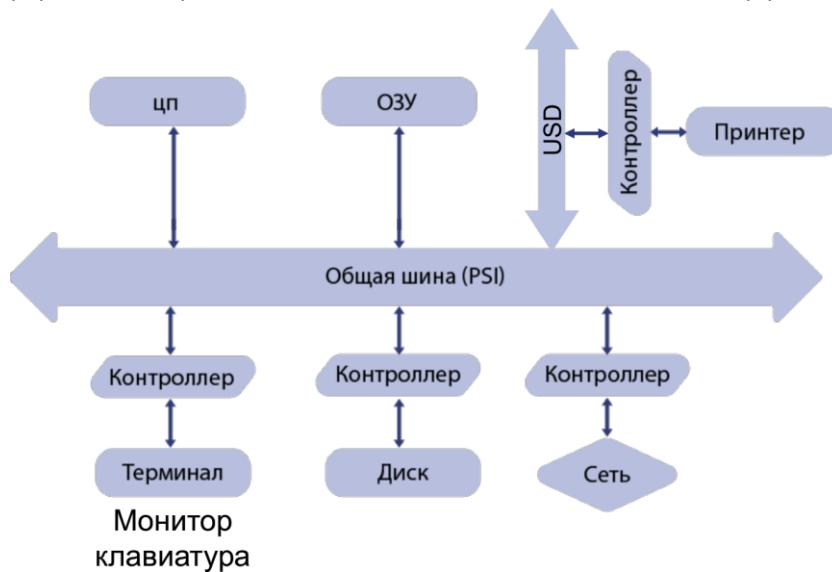
1.1 Понятие «Операционная система». Назначение ОС. Компоненты и функции ОС

Компьютер

По своей сути компьютер — это всего два устройства:

- Вычислитель (центральный процессор, ЦП).
- Оперативная память (ОП или ОЗУ).

Эти два устройства связаны между собой жгутом проводов, обычно называемым шиной или каналом. На шину впаиваются разъемы, в которые можно добавлять другие устройства. ЦП и ОП по сути являются вычислителем и быстрым хранилищем информации. В основном контуре вычислительной системы нужно использовать различное периферийное оборудование, или устройства ввода-вывода информации. На рисунке на общую шину «навешиваются»: монитор, клавиатура, накопитель на магнитных дисках, сетевой адаптер и другие устройства ввода-вывода информации. Чтобы управлять всем этим оборудованием, нужны программы, которые позволили бы синхронизировать доступ по всем устройствам, производить ввод-вывод информации и предоставили бы пользователю какой-то интерфейс для работы со всем этим оборудованием.



Операционная система

Операционная система (ОС) — это набор программ и управляющих структур (таблиц), которые обеспечивают возможность использования аппаратуры компьютера и предоставляют пользователю интерфейс для управления информацией и реализации прикладного и системного ПО.

Назначение операционных систем

По своему назначению ОС можно разбить на три класса:

- **Операционные системы реального времени** — тот набор программ, который должен работать в реальном масштабе времени. Например, самонаводящаяся ракета, запущенная на цель — в головке этой ракеты находится вычислитель (компьютер), который позволяет во время полета координировать действия снаряда в соответствии с внешней обстановкой и не отклоняться от цели. Если эта ОС не будет ОС реального времени, то могут произойти различные инциденты, в т.ч. непопадание в цель.
- **Операционные системы с разделением времени.** Обычные универсальные ОС, такие как Microsoft Windows, UNIX, Linux, Mac OS и т.п. По своему распространению этот класс занимает 70% всего охвата ОС. В таких ОС процессорное время делится между несколькими параллельно работающими процессами, которые конкурируют за ресурсы процессора и ОП.

- **Операционные системы пакетной обработки информации.** Обычно это ОС, предназначенные для обработки большого пакета программ, основная задача которых — считать многочисленный набор программ, который поступает на вход системы.

Функции ОС

- Разделение ресурсов компьютера между параллельно работающими программами.
- Эффективное выполнение операций ввода-вывода (с любого устройства ввода-вывода, которое входит в контур вычислительной системы).
- Предоставление интерфейса пользователю (командная строка, графически интерфейс).
- Восстановление информации и вычислительного процесса в случае ошибок (аварий, сбоев, например, отключение питания).
- Планирование доступа пользователей к общим ресурсам.

ОС управляет следующими ресурсами:

1. Память (оперативная и внешняя (массовая)).
2. Процессорное время (только для некоторых ОС (с распределением времени)).
3. Программы.
4. Периферийное оборудование, устройства ввода-вывода.
5. Данные.

Типы ресурсов:

- Выгружаемые — такие ресурсы можно безболезненно забирать у владеющего ими процесса (например, оперативная память).
- Невыгружаемые — такие ресурсы нельзя забирать у владеющего ими процесса (например, записываемый компакт-диск, принтер, открытый файл пользователя), пока он их сам не освободит.
- Исчерпаемые (память, файл — их всегда можно прочитать до конца, или тонер в принтере).
- Неисчерпаемые (круговой конвейер, когда выход файла заведен на вход файла, что обеспечивает бесконечное чтение).

Последовательность событий, необходимых для использования ресурса, можно представить в следующем виде:

1. Запрос ресурса.
2. Использование.
3. Возврат.

Место ОС в общей системе компьютера

| Прикладные программы | | | | Прикладное ПО | |
|--------------------------------|-----------|--------------|----------------------|------------------------|--|
| Трансляторы | Редакторы | Системы окон | Интерпретатор команд | Операционная система | |
| Ядро ОС | | | | | |
| Система команд (машинный язык) | | | | | |
| Микроархитектура | | | | Аппаратное обеспечение | |
| Физические устройства | | | | | |

| | | | | | |
|---------------------------------------|-----------|-------------|--------------|-----|----------------------|
| Веб-браузер | Matlab | Fine-Reader | Игры | ... | Прикладные программы |
| Трансляторы | Редакторы | Загрузчики | Калькуляторы | ... | Операционная система |
| Ядро операционной системы | | | | | |
| Система команд (машинный язык) | | | | | |
| Микроархитектура | | | | | |
| Физические устройства | | | | | |

Самый нижний уровень содержит физические устройства, состоящие из интегральных микросхем, проводников, источников тока и т.д.

Выше у некоторых компьютеров расположен микроархитектурный (микропрограммный) уровень — примитивная программная прослойка, напрямую работающая с оборудованием и располагающаяся в ПЗУ. Основное его назначение — интерпретация машинных команд. Нужен, чтобы безболезненно переносить программы с одной архитектуры на другую, т.е. сделать компьютер мобильным.

Система команд содержит 50-300 команд, предназначенных для перемещения данных, выполнения арифметических и сдвиговых операций, сравнения величин. Управление устройствами на этом уровне осуществляется с помощью специальных регистров устройств, методов адресации и оперативной памяти компьютера.

Уровень операционной системы находится над аппаратным уровнем и состоит из ядра ОС и системных программ. Ядро — это та часть программ и структуры данных, которые всегда находятся в оперативной памяти. Остальные программы (трансляторы, редакторы, загрузчики) — это тоже системные программы, но они загружаются из внешней памяти в оперативную по мере необходимости.

Уровень прикладных программ. Прикладные программы — приобретаются или пишутся самим пользователем.

ОС взаимодействует с:

- Пользователями.
- Программами (прикладными и системными).
- Аппаратными средствами.

1.2 Понятие процесса ОС. Понятие ядра ОС

Процесс ОС

Процесс — это программа в стадии выполнения вместе с текущими значениями счетчика команд, регистров и переменных.

У каждого процесса есть **адресное пространство** — список адресов в памяти, из которых можно читать и в которые можно писать. Оно содержит саму программу, данные и стек. Как только пользователь запускает программу, ОС формирует запись об этом процессе у себя, куда заносится имя этого процесса, другие атрибуты, связанные с процессом, и сопровождает этот процесс до момента его завершения. В ОС с разделением времени в любой момент могут находиться десятки или даже сотни процессов, которые будут работать одновременно и конкурировать за ресурсы системы.

Каждый процесс состоит как минимум из одного потока.

Квант времени — 100мс (по умолчанию), в течении которых процесс может выполняться. Если за это время программа не завершилась, происходит прерывание по таймеру, процесс ставится в очередь на выполнение. Т.е. по завершению кванта времени ОС вместо данного процесса предоставляет ЦП другому процессу, а данный процесс переносится в список готовых (через какое-то время ОС опять предоставит этому процессу ресурс в виде ЦП).

Состояния процесса

В период своего существования процесс проходит через ряд **дискретных состояний**:

1. Выполняется, если в данный момент времени ему предоставлен ЦП.
2. Готов, если сможет сразу использовать предоставленный ему ЦП.
3. Блокирован, если ожидает наступления некоторого события (например, ввода-вывода информации, завершения другого процесса, закрытия какого-либо файла, срабатывания таймера).
4. (*Приостановлен*) — промежуточное состояние, не называть!

ОС следит за тем, в каком состоянии находится каждый из ее процессов.

Когда в систему поступает некоторое задание, она создает соответствующий процесс, который затем устанавливается в конец списка на выполнение (готовых процессов). Этот процесс постепенно продвигается к головной части списка по мере завершения выполнения предыдущих процессов. Когда процесс окажется первым в списке готовых процессов и когда освобождается ЦП, этому процессу выделяется ЦП, и происходит смена состояния процесса из состояния «Готов» в состояние «Выполняется».

Если же процесс окажется в состоянии блокировки, то ОС может перевести его из состояния «Блокирован» в состояние «Готов» и поставить в список на выполнение.

Чтобы предотвратить либо случайный, либо умышленный монопольный захват ресурсов компьютера каким-то одним процессом, ОС устанавливает в специальном аппаратном таймере прерываний временной интервал (квант времени), в течение которого любому процессу разрешается занимать ЦП.

Если процесс добровольно не освобождает ЦП в течение указанного временного интервала, таймер вырабатывает сигнал прерывания, по которому управление будет передано ОС. После этого ОС переведет ранее выполнявшийся процесс в состояние готовности, а первый процесс из списка готовых — в состояние выполнения.

С каждым процессом связывается его **адресное пространство**: список адресов от минимума до максимума, который процесс может прочесть и в который он может писать. В адресном пространстве содержится сама программа, данные к ней, ее стек.

Со всяким процессом связываются регистры, включая счетчик команд, указатель стека и другие регистры.

Понятие процесса базируется на двух независимых концепциях: **группирования ресурсов и выполнения программ**. С одной стороны у процесса есть адресное пространство, содержащее текст программы, данные, открытые файлы, дочерние процессы и т.д. С другой стороны, процесс можно рассматривать как поток исполняемых команд. У потока есть счетчик команд, регистры, стек и т.д. Хотя поток должен исполняться внутри процесса, следует различать концепции потока и процесса. **Процесс** используется для группирования ресурсов, а **потоки** являются объектами, поочередно исполняющимися на ЦП (т.е. действиями на фоне этих ресурсов).

Концепция потока добавляет процессу возможность выполнения нескольких программ достаточно независимо.

Ядро ОС

Все операции, связанные с процессами, выполняются под управлением той части ОС, которая называется ядром. Ядро представляет собой небольшую часть кода ОС в целом. Однако оно относится к числу наиболее интенсивно используемых компонент системы. По этой причине ядро обычно резидентно размещается в оперативной памяти, в то время как другие части ОС перемещаются во внешнюю память и обратно по мере необходимости.

Ядро ОС — часть кода ОС (набор программ и структур данных), которая всегда находится в оперативной памяти компьютера.

Функции ядра ОС

Ядро — это комплекс процессов (программ), работающих параллельно и реализующих следующие функции:

- **Управление процессами и потоками.** Ядро ОС следит за тем, какие процессы существуют в настоящее время, в каком состоянии они находятся и производит их выполнение до момента завершения.
- **Поддержка операций ввода-вывода.** В основном эту функцию ядра выполняют драйверы УВВ.
- **Распределение и перераспределение оперативной памяти (ОП).** Между одновременно работающими процессами память надо делить — после кванта времени она отнимается у одного процесса и дается другому процессу. ОС следит за тем, какая память свободна и какую память надо забрать у процесса.
- **Управление внешней памятью и поддержка работы файловой системы.** Т.е. это работа с хранилищем данных, которые располагаются во внешней памяти, или работа с файлами.
- **Обработка прерываний.** В любой момент работы компьютера может поступить прерывание. ОС реагирует на это прерывание, обрабатывает его, выполняет действия, предписанные ей при поступлении данного прерывания.
- **Формирование учетной информации** (различная статистическая информация для учета работы компьютера — времени ЦП, которое занимал каждый из процессов, какие задания проставляли, какие процессы блокированы и т.д.). Часть этой информации обычно находится в таблицах, которые ведет ОС. Все эти таблицы находятся в ОП и пользователь может общаться с этими таблицами на уровне функций, которые ему предоставляет ОС, или утилит.
- **Выполнение системных вызовов.**

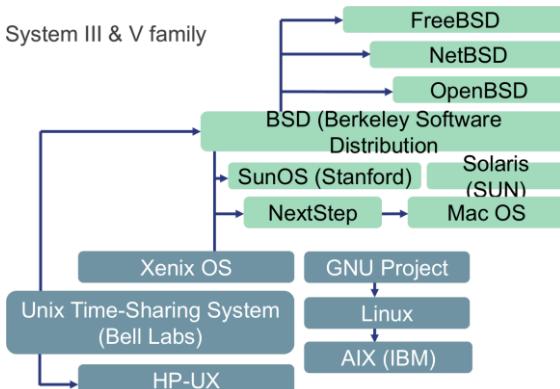
1.3 Операционные системы семейства UNIX

История развития

01.01.1970 — день рождения UNIX. Bell Laboratories, подразделение корпорации AT & T, занимавшейся телекоммуникационным оборудованием.

В настоящий момент, по прошествии 50 лет, практически любая фирма, работающая на рынке вычислительных услуг, имеет в своем арсенале UNIX -подобную ОС.

Эволюция ветки открытого ПО Berkeley Software Distribution или Berkeley Standard Distribution (BSD) и Linux:



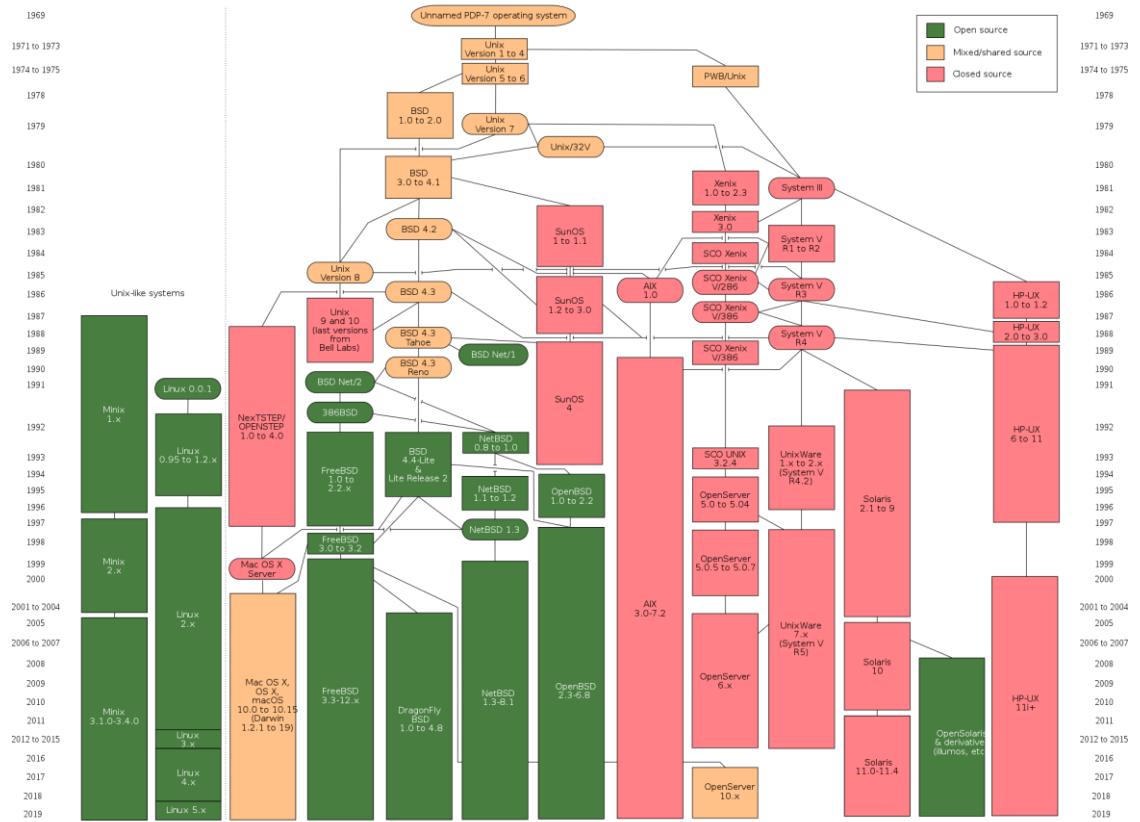
Из недр BSD родились такие ОС, как Mac OS и iOS корпорации Apple, Solaris корпорации SUN.

1990 году — создание UNIX-подобной ОС GNU/Linux.

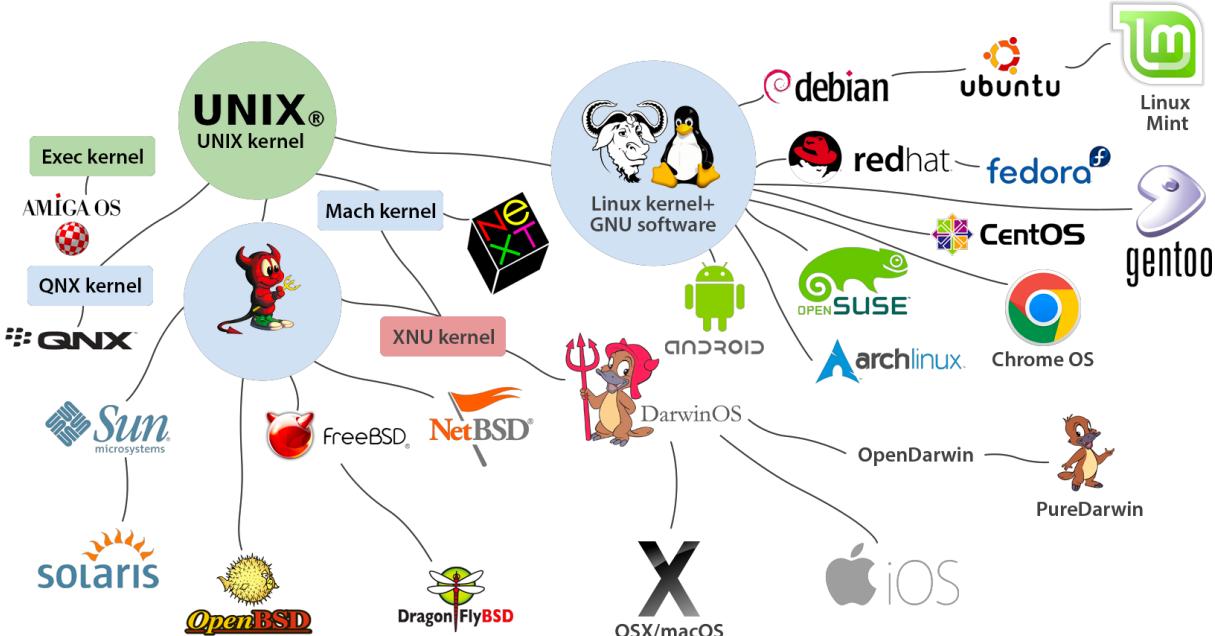
Red Hat Linux легла в основу разработки ОС AIX американской корпорации IBM.

С развитием UNIX-подобных ОС образовалось две ветви — американская System V и европейская POSIX.

Эволюция UNIX и UNIX-подобных систем:



UNIX-подобные системы



UNIX-подобная ОС Android изначально разрабатывалась компанией Android, Inc., которую затем приобрела Google. Основана на ядре Linux и собственной реализации виртуальной машины Java от Google.

[Astra Linux](#) — ОС на базе ядра Linux российского производства, созданная для комплексной защиты информации и построения защищенных автоматизированных систем. Востребована в первую очередь в российских силовых ведомствах, спецслужбах и государственных органах.

Основные характеристики ОС UNIX

ОС реализует единый интерфейс для обработки информации. Все ресурсы ОС унифицированы. Устройства ввода-вывода, память, механизмы синхронизации и др. — ассоциированы с понятием файла. Т.е., например, работа с любым устройством ввода-вывода или с каналами для ввода-вывода информации, или с семафорами рассматривается, с точки зрения пользователя, как работа с обычным файлом.

Все работы в ОС UNIX представлены множеством конкурирующих процессов/потоков. Логически каждый процесс выполняется своим виртуальным процессором в своем виртуальном адресном пространстве.

Достоинства UNIX систем

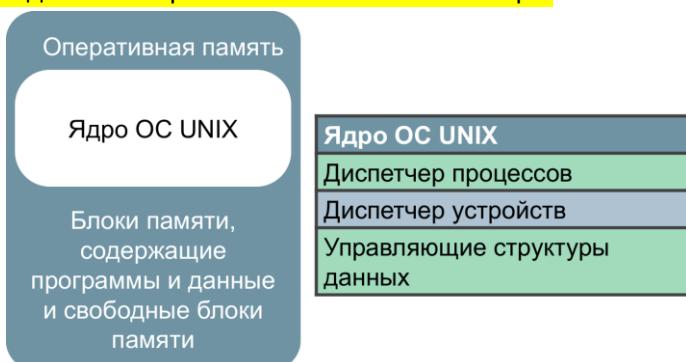
- **Открытость исходного кода.**
- **Инструментальность** (насыщенность программными продуктами) — набор программных средств, облегчающих конструирование прикладного ПО. Столько программ, сколько сделано под ОС семейства UNIX, не сделано ни под одну ОС, существующую на данный момент. Любые программные пакеты обычно разрабатываются под UNIX-подобную ОС, а затем уже разрабатываются под другую ОС.
- **Мобильность** — возможность переноса ОС на вычислительные системы с другой архитектурой с минимальными затратами.
- **Сетевая направленность** (протоколы TCP/IP, UPD, FTP и др. — это все родные протоколы UNIX-систем).
- **Многозадачность** — поддержка одновременного выполнения нескольких программ.
- **Многопользовательность** — поддержка одновременного присутствия нескольких пользователей.

Недостатки UNIX систем

- Не поддерживается режим реального времени, т.е. доступ к векторам прерывания закрыт.
- Снижение эффективности при решении однотипных задач.
- Слабая устойчивость к аппаратным сбоям.

1.4 Ядро операционных систем семейства UNIX

Как уже говорилось выше, **ядро ОС** — часть кода ОС (набор программ и структур данных), которая всегда находится в оперативной памяти компьютера.



Ядро ОС UNIX состоит из двух секций:

- **Секция управляющих структур** (табличная секция).
- **Программная секция** (программы).

В секциях управляющих структур располагаются таблицы ядра — таблицы процессов и потоков, файлов, индексных дескрипторов файлов, связи драйверов, семафоров и т.д. (всего около 40).

Программная секция состоит из двух основных частей:

- **Диспетчер процессов и потоков** (машинно-независимая часть).
- **Диспетчер устройств ввода-вывода** (машинно-зависимая часть).

Секция управляющих структур (табличная секция)

Помимо программ ОС ведет таблицы. Каждая запись в любой таблице обычно называется дескриптором таблицы и в эту запись записываются атрибуты того или иного ресурса, которым манипулирует ОС.

К **секции управляющих структур** относятся различные **таблицы**, которые ОС формирует/сопровождает:

1. **Таблица процессов** (каждая запись таблицы характеризует один процесс, как только процесс завершается, эта запись из таблицы удаляется).
2. **Таблица файлов** (содержит информацию обо всех открытых файлах в ОС).
3. **Таблицы связей драйверов** (специальные таблицы, которые помогают перенастраиваться на список и состав внешних устройств).
4. **Таблица очередей (каналов) сообщений** (отслеживает все очереди сообщений, которые существуют на данный момент в ОС).
5. **Карта памяти** (таблица, в которой отражается распределение и перераспределение ОП).
6. **Таблица наборов семафоров и др.**

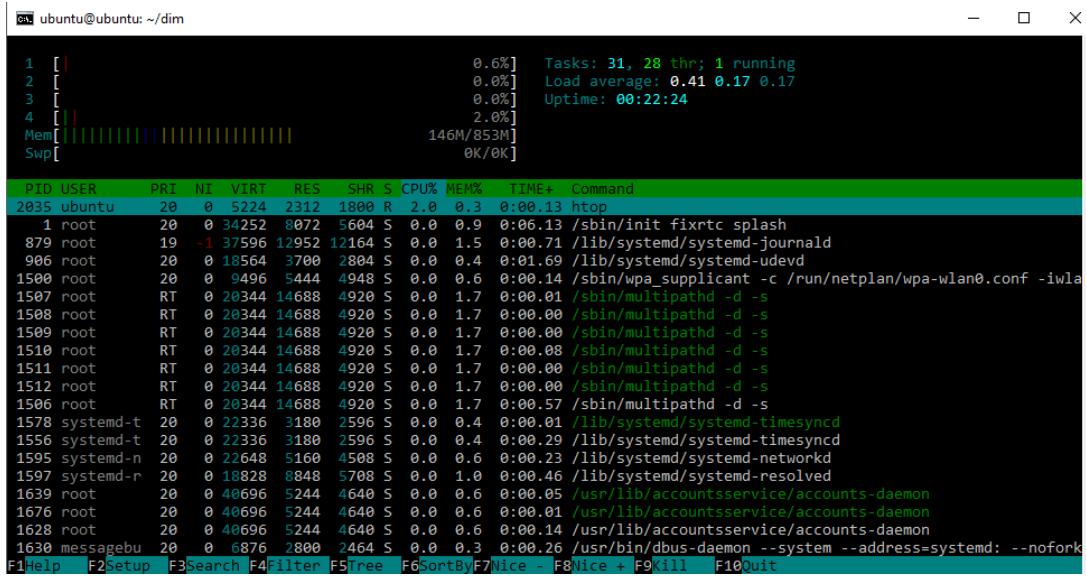
Всего около 40 таблиц. Таблицы находятся постоянно в ОП и ОС постоянно следят за состоянием этих таблиц, она формирует новые и удаляет старые записи.

Программная секция ядра. Диспетчер процессов

Диспетчер процессов и потоков — это машинно-независимая часть кода ядра ОС, которая может переноситься без существенных изменений с одной вычислительной платформы на другую.

Диспетчер процессов — набор программ, который выполняет следующие функции:

1. Определяет последовательность выполнения процессов/потоков.
2. Распределяет ресурсы системы, в т.ч. оперативную память.
3. Осуществляет обработку системных вызовов.
4. Обрабатывает сигналы.



The screenshot shows the output of the htop command on an Ubuntu system. At the top, it displays system statistics: Tasks: 31, 28 thr; 1 running, 0.0% Load average: 0.41 0.17 0.17, Uptime: 00:22:24. Below this, memory usage is shown: Mem[146M/853M] 2.0%, Swap[0K/0K]. The main part of the screen is a table of processes. The columns are: PID, USER, PRI, NI, VIRT, RES, SHR, S, CPU%, MEM%, TIME+, and Command. The table lists various processes including root, systemd, and user applications like splash, systemd-journal, wpa_supplicant, and accountsservice.

| PID | USER | PRI | NI | VIRT | RES | SHR | S | CPU% | MEM% | TIME+ | Command |
|------|------------|-----|----|-------|-------|-------|---|------|------|---------|-----------------------------------------------------------|
| 2035 | ubuntu | 20 | 0 | 5224 | 2312 | 1800 | R | 2.0 | 0.3 | 0:00.13 | htop |
| 1 | root | 20 | 0 | 34252 | 8072 | 5604 | S | 0.0 | 0.9 | 0:06.13 | /sbin/init fixrtc splash |
| 879 | root | 19 | -1 | 37596 | 12952 | 12164 | S | 0.0 | 1.5 | 0:00.71 | /lib/systemd/systemd-journald |
| 966 | root | 20 | 0 | 18564 | 3700 | 2804 | S | 0.0 | 0.4 | 0:01.69 | /lib/systemd/systemd-udevd |
| 1500 | root | 20 | 0 | 9496 | 5444 | 4948 | S | 0.0 | 0.6 | 0:00.14 | /sbin/wpa_supplicant -c /run/netplan/wpa-wlan0.conf -iwla |
| 1507 | root | RT | 0 | 20344 | 14688 | 4920 | S | 0.0 | 1.7 | 0:00.01 | /sbin/multipathd -d -s |
| 1508 | root | RT | 0 | 20344 | 14688 | 4920 | S | 0.0 | 1.7 | 0:00.00 | /sbin/multipathd -d -s |
| 1509 | root | RT | 0 | 20344 | 14688 | 4920 | S | 0.0 | 1.7 | 0:00.00 | /sbin/multipathd -d -s |
| 1510 | root | RT | 0 | 20344 | 14688 | 4920 | S | 0.0 | 1.7 | 0:00.08 | /sbin/multipathd -d -s |
| 1511 | root | RT | 0 | 20344 | 14688 | 4920 | S | 0.0 | 1.7 | 0:00.00 | /sbin/multipathd -d -s |
| 1512 | root | RT | 0 | 20344 | 14688 | 4920 | S | 0.0 | 1.7 | 0:00.00 | /sbin/multipathd -d -s |
| 1506 | root | RT | 0 | 20344 | 14688 | 4920 | S | 0.0 | 1.7 | 0:00.57 | /sbin/multipathd -d -s |
| 1578 | systemd-t | 20 | 0 | 22336 | 3180 | 2596 | S | 0.0 | 0.4 | 0:00.01 | /lib/systemd/systemd-timesyncd |
| 1556 | systemd-t | 20 | 0 | 22336 | 3180 | 2596 | S | 0.0 | 0.4 | 0:00.29 | /lib/systemd/systemd-timesyncd |
| 1595 | systemd-n | 20 | 0 | 22648 | 5160 | 4508 | S | 0.0 | 0.6 | 0:00.23 | /lib/systemd/systemd-networkd |
| 1597 | systemd-r | 20 | 0 | 18828 | 8848 | 5708 | S | 0.0 | 1.0 | 0:00.46 | /lib/systemd/systemd-resolved |
| 1639 | root | 20 | 0 | 40696 | 5244 | 4640 | S | 0.0 | 0.6 | 0:00.05 | /usr/lib/accountsservice/accounts-daemon |
| 1676 | root | 20 | 0 | 40696 | 5244 | 4640 | S | 0.0 | 0.6 | 0:00.01 | /usr/lib/accountsservice/accounts-daemon |
| 1628 | root | 20 | 0 | 40696 | 5244 | 4640 | S | 0.0 | 0.6 | 0:00.14 | /usr/lib/accountsservice/accounts-daemon |
| 1630 | messagebus | 20 | 0 | 6876 | 2800 | 2464 | S | 0.0 | 0.3 | 0:00.26 | /usr/bin/dbus-daemon --system --address=systemd: --nofork |

Программная секция ядра. Диспетчер УВВ

Диспетчер внешних устройств контролирует и обеспечивает передачу информации между ОП компьютера и УВВ.

В него входят драйверы символьных (байтовых), сетевых, дисковых устройств, дисциплины линий (канал обмена информацией между компьютером и внешним устройством), буферный и страничный кэши, маршрутизаторы, сетевые протоколы, файловые системы, виртуальная память и др.

Вычислительные системы по своему составу (аппаратному) отличаются друг от друга. **Диспетчер УВВ** — это машинно-зависимая часть кода ядра ОС, которая будет существенно отличаться при переносе с одной вычислительной платформы на другую. В целом, это не большой недостаток, т.к. драйверы УВВ обычно поставляются вместе с устройством и их достаточно легко установить в рамках ОС.

```
root@192:~# lsmod | head -20
Module           Size  Used by
ctr              16384  2
ccm              20480  6
rtl8xxxu        131072  0
binfmt_misc     24576  1
nls_ascii        16384  1
nls_cp437        20480  1
vfat             20480  1
fat              86016  1 vfat
snd_sof_pci      20480  0
snd_sof_intel_hda_common  90112  1 snd_sof_pci
snd_hda_codec_hdmi   73728  1
snd_sof_intel_hda    20480  1 snd_sof_intel_hda_common
snd_sof_intel_byt    20480  1 snd_sof_pci
snd_sof_intel_ipc    20480  1 snd_sof_intel_byt
snd_sof            126976  4 snd_sof_pci,snd_sof_intel_hda_common,snd_sof_intel_byt,snd_sof_intel_ipc
snd_sof_xtensa_dsp   16384  2 snd_sof_intel_hda_common,snd_sof_intel_byt
btusb             57344  0
btrtl             24576  1 btusb
btbcm             20480  1 btusb
```

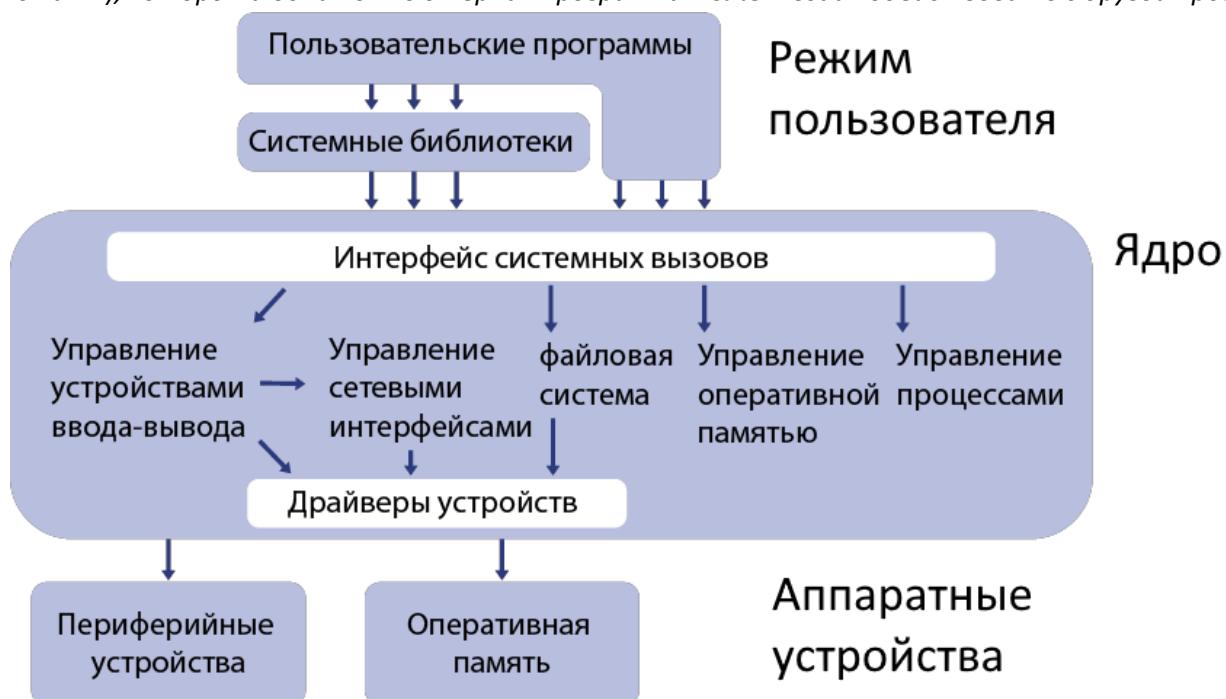
1.5 Понятие системного вызова (СВ). Особенности применения СВ в UNIX

Системный вызов

Системный вызов — это интерфейс между прикладными программами и ядром ОС. СВ позволяют прикладным программам общаться с остальной частью системы. Аппарат системных вызовов используется когда во время выполнения программы необходимо выполнить другую программу или предоставить пользовательскому процессу системные функции.

Каждая ОС характеризуется функциями (набором СВ), которые она предоставляет пользователю для манипулирования ее ресурсами. **Набор функций, которые предоставляет ОС, обычно называется системными вызовами.** В рамках UNIX-подобных ОС системных вызовов порядка 1000. В рамках ОС Windows системных вызовов уже 13000.

Существенным достоинством UNIX-подобных ОС является то, что пользователь при разработке своих программных продуктов может напрямую обращаться к системным вызовам. В ОС Windows такого уже сделать нельзя, т.к. обращение к СВ там идет через API. **API (application programming interface, интерфейс прикладного программирования)** — описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой.



Стандарты на СВ в UNIX-подобных системах

Для того, чтобы программы, разработанные под одну UNIX-систему, были переносимые и работали под любой UNIX-подобной ОС, разработчики сформировали стандарты интерфейсов ОС.

Стандарт интерфейсов ОС — это набор формальных синтаксических (интерфейсных) и семантических (поведенческих) правил специфицируемых средств операционной системы. Т.е. функция должна принимать одни и те же аргументы, должна возвращать одни и те же значения и должна производить одни и те же действия.

Стандарты:

- System V Interface Definition (SVID) — «американский» стандарт, описывающий поведение ОС UNIX System V компании AT&T, включая набор системных и библиотечных вызовов, приложения и устройства.
- POSIX.1 (Portable Operating System Interface 1003.1 — переносимый интерфейс операционных систем) — «европейский» стандарт.
- X/Open Portability Guide (XPG).
- ANSI C.
- AES Application Environment Specification (AES) (обобщающий).

Два основных стандарта — System V и POSIX.1.

Примеры системных вызовов

Синтаксически применение СВ похоже на вызов процедуры или функции (*подпрограммы*), однако исполняемый код системного вызова находится в ядре ОС (постоянно загружен в ОП компьютера), *а не в загрузочном модуле*.

int open (const char * filename, int flags, mode_t mode) — открывает файл в заданном режиме открытия (чтение, запись, чтение и запись и т.п.) или создает новый файл;

int wait (int * account) — позволяет приостановить процессу-родителю свое выполнение до завершения процесса-сына;

int read (int fd, char* buf, int n) — считывает из открытого файла указанное количество байтов в ОП компьютера. Это единственная функция чтения, которая работает в системе без буферизации;

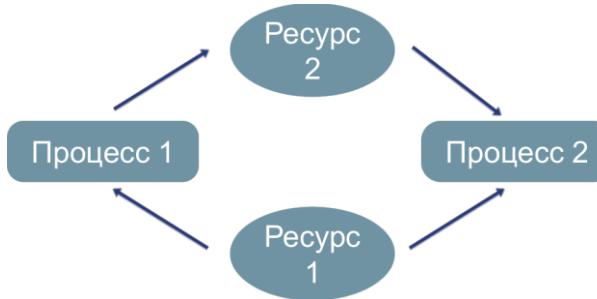
int execl (const char *full_file_name, char * arg[0], char *arg[1], ..., char * (0)) — вызывает новую программу, вместо уже выполняющейся без возврата в вызывающую программу (запускает другой процесс в рамках одного процесса с заменой контекста, в рамках которого был запущен новый процесс);

int pipe (int * p) — создает коммуникационный канал межпроцессной связи между двумя взаимосвязанными параллельно работающими процессами.

2. Процессы и потоки в ОС UNIX

2.1 Понятия, связанные с процессом

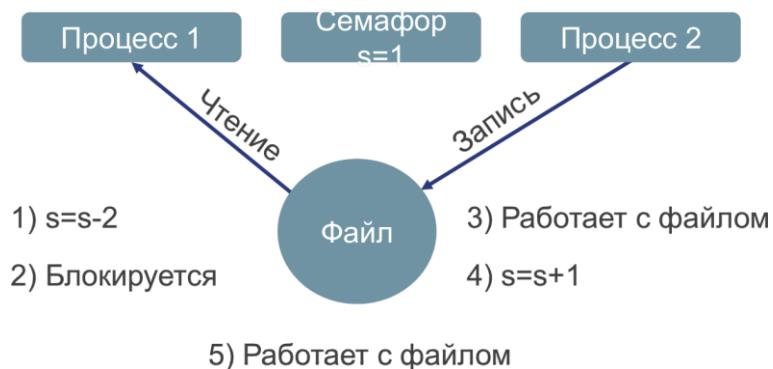
- **Параллельные процессы** — это две программы, работающие одновременно.
- **Асинхронные процессы** — это параллельные процессы, работающие независимо друг от друга.
- **Взаимоисключающие процессы** — это параллельные процессы, использующие общий невыгружаемый ресурс. Тот процесс, который захватил этот ресурс, будет замедлять/блокировать работу других процессов до тех пор, пока он не освободит этот ресурс.
- **Задание** — это группа процессов, работающих как единое целое для достижения какой-то заданной цели. Т.е. если надо решить какую-то задачу с помощью нескольких параллельно работающих процессов, то их можно объединить в задание и это задание будет выполняться до тех пор, пока каждый из процессов, составляющих задание, не закончит свое выполнение.
- **Тупиковый процесс** — это группа процессов, ожидающих какого-то события, которое никогда не произойдет. Рассмотрим пример на рисунке. Процессу 1 выделен ресурс 1, а процессу 2 выделен ресурс 2. В это время процесс 1 запрашивает ресурс 2, а процесс 2 запрашивает ресурс 1. Таким образом эти процессы попадают в тупиковую ситуацию, из которой обычно нет выхода. Т.е. кто-то должен освободить этот ресурс, чтобы выйти из этой ситуации. Обычно, если в процессе работы ОС возникает много тупиковых ситуаций, то пользователи говорят, что у них «зависла» ОС.



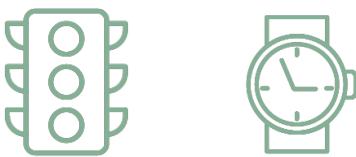
Инструменты синхронизации процессов и потоков

- **Семафор** — системная переменная, которая является объектом ядра ОС и используется для синхронизации процессов. Принимает неотрицательные целочисленные значения. Значение семафора не может быть отрицательным.
 - **Семафоры** — общесистемные переменные, находящиеся в ядре ОС (эти переменные видят все процессы), которые могут модифицироваться и использоваться процессами, запущенными на одном компьютере, для синхронизации их выполнения. Семафоры обычно используются в сочетании с разделяемой памятью для управления доступом к данным, находящимся в той или иной области разделяемой памяти.
- Рассмотрим на примере, по какому принципу с помощью семафоров осуществляется синхронизация действий параллельных процессов. Например, два процесса работают с общим ресурсом — файлом. Один процесс должен записать информацию в файл, а другой процесс должен прочитать информацию из файла и эти действия надо синхронизировать. Т.е. пока процесс 2 не запишет информацию, процесс 1 не должен ее читать. Синхронизация осуществляется с помощью семафора. Если значение семафора равно единице, то для процесса 1, который должен считать из файла информацию, значение семафора уменьшается на 2. Т.к. значение семафора не может быть отрицательным ($1-2 = -1$), то ОС блокирует процесс 1, пытающийся это сделать на этом действии, и процесс будет ждать до тех пор, пока кто-то не увеличит значение семафора, чтобы результат вычитания был неотрицательным (чтобы можно было вычесть двойку). В это время процесс 2 осуществляет запись в файл и после того, как информация в файл запишется, этот

процесс изменяет значение семафора (добавляет единицу) и отпускает процесс 1, который должен читать информацию из файла.



- **Мьютекс / mutex** — сокращенная версия семафора. Как и семафор, мьютекс — общесистемные переменные, находящиеся в ядре ОС (эти переменные видят все процессы). В отличии от семафора, mutex «не умеет считать» и принимает только два значения — 0 или 1 (заблокировано или разблокировано). Если с помощью одного семафора можно синхронизировать бесконечное множество процессов, то с помощью мьютекса это множество ограничено.
- **Критическая секция / critical section** — очень похожа на мьютекс, т.к. принимает всего два значения — 0 или 1 (заблокировано или разблокировано), но критическая секция **используется для синхронизации выполнения потоков в рамках одного процесса**, т.к. находится в адресном пространстве процесса и не является объектом ядра ОС.
- **События / event** бывают двух типов: сигнализирующие (сбрасываемые вручную) и не сигнализирующие (сбрасываемые автоматически). Например: подождать, пока не закончится выполнение какого-то процесса, подождать, пока не сработает таймер, подождать, пока не завершится ввод-вывод и т.д. Так же как и мьютексы, события характеризуются двумя значениями — 0 или 1 (свершилось событие или не свершилось)
- **Таймеры / timer** — синхронизация параллельно работающих процессов осуществляется за счет часов. Т.е. можно синхронизировать процессы по времени. Например, в 2 часа один процесс должен что-то сделать, а в 3 часа другой процесс должен что-то сделать.



Понятия, связанные с потоком

У любого процесса есть хотя бы один **поток управления** (последовательность команд для выполнения конкретного действия) и один счетчик команд.

В некоторых современных ОС реализована возможность создания нескольких потоков управления в одном процессе. Такие потоки получили названия **нитей (threads)** или **легковесных процессов** или просто **потоками**. В ходе выполнения процесса могут создаваться другие потоки. Например вычисление разных функций $\sin(x)$ и $\ln(x)$ от одного аргумента x можно реализовать в разных потоках для того, чтобы они считались одновременно. Т.е. **поток** — это набор команд, который можно выполнить независимо от набора других команд в рамках одного процесса.

У всех потоков процесса одно общее адресное пространство оперативной памяти и общие ресурсы, но у каждой — личные счетчик команд, регистры, приоритеты и состояния. Информация о потоках записывается в таблицу, которую ведет ядро ОС. Каждая ее запись называется дескриптором потока и содержит атрибуты одного потока. У каждого потока может быть состояние готов, выполняется или блокирован и у каждого

потока появляется приоритет. Приоритет рассчитывается по приблизительно такой же формуле, как и для процесса, только теперь приоритеты присутствуют не у процессов, а у потоков. Соответственно, планирование в ОС будет производиться уже не на уровне процессов, а на уровне потоков.

Понятие процесса базируется на двух независимых концепциях: группирования ресурсов и выполнения программ. С одной стороны у процесса есть адресное пространство, содержащее текст программы, данные, открытые файлы, дочерние процессы и т.д. С другой стороны, процесс можно рассматривать как поток исполняемых команд. У потока есть счетчик команд, регистры, стек и т.д. Хотя поток должен исполняться внутри процесса, следует различать концепции потока и процесса. Процесс используется для группирования ресурсов, а потоки являются объектами, поочередно исполняющимися на ЦП (т.е. действиями на фоне этих ресурсов). У потоков общие файлы, общие УВВ и другие ресурсы.

Концепция потока добавляет процессу возможность выполнения нескольких программ достаточно независимо.

Волокно / fiber — поток, выполняемый в пространстве пользователя, т.е. не обращается к ядру ОС. Такой поток выполняется быстрее. Применяется исключительно в ОС Windows.

2.2 Дескриптор процесса

При создании процесса ядро ОС формирует дескриптор процесса (блок управления процессом). Это структура данных, в которой содержатся все атрибуты созданного процесса. Каждый дескриптор процесса описывает свой уникальный процесс.

Ядро ОС ведет таблицу процессов, каждая запись в которой является дескриптором одного процесса. Пока процесс выполняется, запись о нем есть в таблице дескрипторов процессов. Как только процесс завершился, запись о нем удаляется из таблицы.

Таблица дескрипторов процессов

| Номер | Идентификатор | Приоритет | Состояние | Указатели адресов памяти | Указатели выделенных ресурсов | Машинные регистры |
|-------|---------------|-----------|-----------|--------------------------|-------------------------------|-------------------|
| 0 | ... | ... | ... | ... | ... | ... |
| 1 | ... | ... | ... | ... | ... | ... |
| 2 | ... | ... | ... | ... | ... | ... |

Типовой дескриптор процесса

Типовой дескриптор процесса — структура данных, содержащая следующие поля (атрибуты):

1. **Идентификатор (имя) процесса** — число PID.
2. **Приоритет** — целое число. Чем меньше это число, тем выше приоритет у процесса. Если число отрицательное, то процесс с очень высоким приоритетом. Если ОС поддерживает потоки, то приоритет оказывается у потоков, но они будут формироваться по той же схеме и характеризоваться теми же числами.
3. **Текущее состояние** — готов, выполняется, блокирован. Если ОС поддерживает потоки, то этот атрибут переходит в ведение потоков.
4. **Указатели адресов памяти, которые занимает процесс.** Те блоки (страницы) памяти, в которые загружается данный процесс.
5. **Указатели выделенных ресурсов.** ОС регулярно следит за теми ресурсами, которые принадлежат процессу, потому что невыгружаемые ресурсы могут потребоваться другим процессам, и нужно знать, когда же этот ресурс будет освобожден. (Выгружаемые ресурсы можно безболезненно забирать у владеющего ими процесса, а невыгружаемые нельзя забирать у владеющего ими процесса, пока он их сам не освободит).

6. **Машинные регистры** (область сохранения машинных регистров) нужны для того, чтобы процесс мог сохранять свое состояние. Процесс работает квантами времени (по умолчанию 100 мс). Если за это отведенное время процесс успел выполнить свою работу, то ОС его завершит. В противном случае процесс, используя машинные регистры, должен сохранить свое состояние для того, чтобы в следующий момент, когда ему выделится ЦП, начать свое выполнение именно с того состояния, на котором он завершился на предыдущем этапе.

Машинные регистры

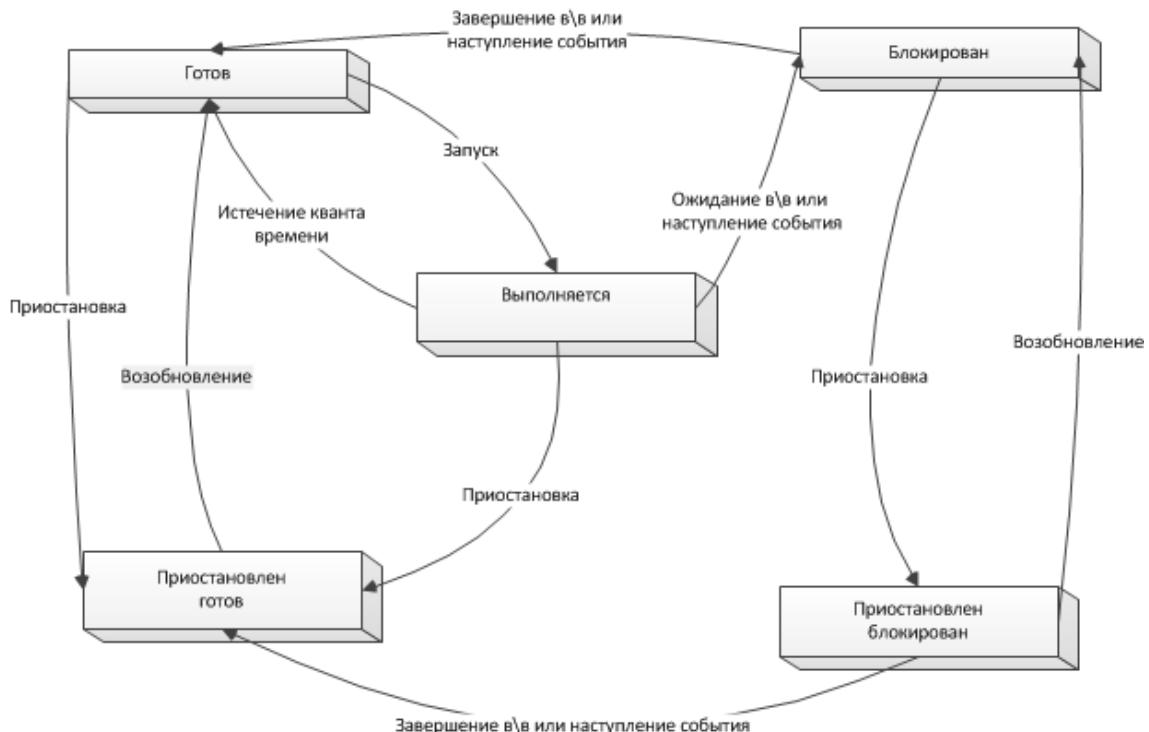
Когда ОС переключает ЦП с процесса на процесс, она использует машинные регистры, предусмотренные в дескрипторе процесса, чтобы запомнить информацию, необходимую для рестарта каждого процесса.

Операции над процессами

ОС поддерживает следующие **операции над процессами**:

1. **Создание или образование процесса** — состоит из множества операций:
 - a. Присвоение имени (идентификатора) процессу — числа PID.
 - b. Формирование дескриптора процесса.
 - c. Определение начального приоритета процесса.
 - d. Выделение процессу требуемых ресурсов. Если нужных ресурсов нет, то процесс создастся и будет в состоянии блокировки, пока ресурс не освободится.
 - e. Занесение процесса в список процессов с состоянием «готов».
2. **Уничтожение** (с помощью СВ `_exit()` или с помощью сигналов (сигнал — программная версия аппаратных прерываний)):
 - a. Завершение процесса.
 - b. Удаление записи из таблицы дескрипторов процессов.
3. **Изменение приоритета** происходит постоянно для каждого процесса, функционирующего в системе. Во время своего создания процесс получает начальный приоритет, в процессе своего выполнения приоритет процесса меняется (пересчитывается 1 раз в секунду). Приоритет процесса падает со временем (число приоритета в ходе выполнения процесса постоянно увеличивается, если не происходит каких-то неординарных событий).
4. **Возобновление** (`restart`) происходит после того, как закончился квант времени и ранее приостановленный процесс снова получает свое распоряжение ЦП:
 - a. Восстановление образа процесса с помощью машинных регистров.
5. **Блокирование** происходит во время операций ввода-вывода, либо при выполнении СВ, когда процессу приходится обращаться в область памяти ядра, либо во время ожидания каких-либо событий (срабатывание таймера по СВ `sleep()`, ожидание завершения параллельного процесса и т.п.).
6. **Пробуждение** (перевод процесса из состояния «блокирован» в состояние — «готов») после какого-то временного интервала, связано с работой таймера.
7. **Запуск** (предоставление процессу ЦП) — функция планировщика, который выбирает из очереди готовый к выполнению процесс с наивысшим приоритетом:
 - a. Сортировка очереди готовых к выполнению процессов по приоритетам.

+Схема состояний процесса



Приостановленный процесс состоит из собственного адресного пространства, обычно называемого образом памяти (*core-файл*) и дескриптора.

2.3 Процессы в UNIX-подобных ОС

Структура процесса

Структура процесса в момент своего нахождения в ОП представляется следующими сегментами:

1. **Процедурный сегмент**: машинные команды и константы.
2. **Сегмент данных**: данные, инициированные при компиляции, которые могут изменяться в процессе работы.
3. **Динамический сегмент**: данные, не инициированные при компиляции. Выделяется при загрузке исполняемого файла в ОП. Содержит пустую память.
4. **Контекст процесса** — таблица, в которой содержится системная информация о процессе при его нахождении в ОП. Эта таблица является составляющей процесса и в ней содержится 5 полей. Контекст процесса не относится к адресному пространству процесса, в отличии от процедурного сегмента, сегмента данных и динамического сегмента. Доступ к контексту процесса имеет только ядро ОС.

Область контекста не относится к адресному пространству процесса, однако контекст подвергается свопингу (выгрузке из ОП во внешнюю память) совместно с процедурным сегментом и сегментом данных.

Область контекста не относится также и к ядру ОС, т.к. вместе с процессом часто «покидает» ОП.

Динамический сегмент содержит пустую память и выгружать его нет смысла, т.к. в следующий раз, когда процесс получит в свое распоряжение ЦП, эта пустая память снова будет выделена процессу.

Оперативная память

| Процедурный сегмент 1 | Процедурный сегмент 2 | Свободно |
|------------------------|------------------------|----------|
| Сегмент данных 1 | Сегмент данных 2 | |
| Динамический сегмент 1 | Динамический сегмент 2 | |
| Контекст процесса 1 | Контекст процесса 2 | |

Примеры того, как данные разносятся по сегментам

```

/* Под целочисленные переменные i и g выделяется по 4 байта.
Под переменную i просто выделяется 4 байта, они будут
пустыми до тех пор, пока переменная i не получит какое-то
значение. Под переменную g также выделяется 4 байта и туда
записывается число 10.
Соответственно, при распределении этих переменных в памяти,
переменная i попадет в динамический сегмент, а переменная g -
в сегмент данных. */
int i, g = 10;
/* Под символьные массивы a и b выделяется по 50 байт,
Однако массив a попадет в динамический сегмент, а массив b
попадет в сегмент данных, потому что у него данные
принициализированы: */
char a[50], b[50] = "Привет";

// Динамический сегмент (байты памяти выделены, но они пустые):
int i;
char a;

// Сегмент данных:
int g = 10;
char b[50] = "Привет";

```

Итак, **процедурный сегмент, сегмент данных и динамический сегмент** формируют **адресное пространство процесса**, а **контекст процесса** не относится к адресному пространству процесса — это просто **таблица**. Подвергаются свопингу контекст процесса, процедурный сегмент и сегмент данных.

Процедурный сегмент и сегмент данных составляют область процесса, которая формируется при трансляции.

2.4 Дескриптор процесса в ОС UNIX. Контекст процесса в ОС UNIX

Контекст процесса

Контекст процесса — структура данных (табличная запись), содержащая информацию о процессе при его нахождении в ОП и включающая 5 полей:

1. **Машинные регистры**. В них сохраняется информация о текущем состоянии процесса.
2. **Таблицу пользовательских дескрипторов файлов, открытых процессом (ТПДОФ** процесса). В этом поле отображены все файлы, которые открыл данный процесс.
3. **Информацию о состоянии текущего системного вызова**, включающую информацию о состоянии СВ, его параметры и результаты. Если в текущий момент процесс не выполняет СВ, это поле остается пустым.
4. **Учетную информацию** (указатель на таблицу, учитывающую процессорное время, использованное процессом).
5. **Стек ядра** (фиксированный стек для работы процесса в режиме ядра).

Область контекста не относится к адресному пространству процесса, однако контекст подвергается свопингу совместно с процедурным сегментом и сегментом данных. Область контекста не относится также и к ядру ОС, т.к. вместе с процессом часто «покидает» ОП. Доступ к контексту процесса имеет только ядро ОС.

Дескриптор процесса

Кроме контекстов процессов, ядро поддерживает также таблицу процессов, являющуюся резидентной (т.е. постоянно находящейся в ОП), в которой отслеживаются все активные процессы. Некоторые из них инициируются ядром и являются системными процессами, остальные связаны с пользователями, называются пользовательскими процессами.

Дескриптор процесса — объект ядра ОС. Как мы уже знаем, ядро ОС ведет таблицу процессов, каждая запись в которой является дескриптором одного процесса. Пока процесс выполняется, запись о нем есть в таблице дескрипторов процессов. Как только процесс завершился, запись о нем удаляется из таблицы.

Итак, в таблице процессов содержатся дескрипторы процессов.

Дескриптор процесса в ОС UNIX, в отличие от типового дескриптора процесса, содержит намного больше атрибутов — 16 полей. Для сравнения, дескриптор процесса в ОС Windows содержит 18 полей.

Дескриптор процесса в ОС UNIX содержит следующую информацию (поля):

1. **PID** — идентификатор (имя) процесса — число.
2. **Текущее состояние процесса**. У процесса может быть три состояния: **готов**, **выполняется**, **блокирован**.
3. **PPID** — идентификатор родительского процесса. Все процессы в UNIX-подобных ОС создаются при помощи СВ **fork()** и имеют родителя (тот процесс, который создал текущий процесс). Исключение составляет процесс с PID = 0, который создается по другой схеме.
4. **UID** — идентификатор владельца процесса (**пользователя**). У каждого процесса в UNIX-подобных ОС есть владелец.
5. **GID** — идентификатор группы (**пользователей**). Ряд пользователей, которые могут общаться с данным процессом.
6. **Время до истечения интервала будильника**. Время, оставшееся до пробуждения данного процесса, если используется СВ **alarm()** или какие-то функции, связанные с временными показателями, например, **sleep()**.
7. **События, которые процесс ожидает (если есть)**. Нужно для того, чтобы ОС вовремя переводила процесс из состояния «блокирован» в состояние «готов».
8. **Сигнальная маска** — массив сигнальных флагов, отражающая, какие сигналы процесс ожидает. Мaska указывает, какие сигналы (сигнал — программная версия аппаратных прерываний и все аппаратные прерывания реализованы в виде сигналов) игнорируются, какие перехватываются и какие заблокированы. Если в сигнальной маске бит нулевой, то процесс игнорирует данный сигнал. Если же в сигнальной маске указан единичный бит, что процесс ждет этот сигнал и должен обязательно на него прореагировать.
9. **Приоритет CPU**. Общее число тиков таймера, которое произошло с момента начала работы процесса (счетчик времени, в течении которого работал процесс). Системная составляющая приоритета процесса. Каждый процессор оснащен кристалловым генератором, испускающим тики с определенной заданной частотой. Зафиксированное за выделенный процессу время количество тиков будет увеличивать данную компоненту приоритета процесса.
10. **Приоритет NICE**. Пользовательская составляющая приоритета процесса. Связана с решением пользователя понизить или повысить приоритет процесса. Этую компоненту приоритета процесса пользователь меняет в своих программах вызывая СВ **nice()**. Компонента NICE может изменяться в интервале от -20 до +20.
11. **Приоритет PRI** (если ОС поддерживает потоки, то приоритеты уходят к потокам, а в дескрипторе процесса этого поля не будет). Приоритет — целое число. Чем меньше это число, тем выше приоритет у процесса. Если число отрицательное, то процесс с очень высоким приоритетом.
12. **Счетчик использования процессорного времени**. Время нахождения процесса в ОП или в области свопинга.

- 13. Указатель на процедурный сегмент.** Если используется страничная организация памяти (виртуальная память), то указатель на таблицу страниц процедурного сегмента.
- 14. Указатель на сегмент данных.** Если используется страничная организация памяти, то указатель на таблицу страниц сегмента данных.
- 15. Указатель на динамический сегмент.** Если страничная организация памяти, то указатель на таблицу страниц динамического сегмента.
- 16. Указатели внешней памяти.** Указатели на расположение образа процесса во внешней памяти (список связанных блоков памяти), если его нет в ОП.

Как видно, некоторые поля дескриптора процесса в OS UNIX можно условно сгруппировать:

- (с 9 по 11) — Приоритет процесса характеризуют целых 3 атрибута (компоненты) — CPU, NICE и PRI.
- (с 13 по 15) — Указатели на процедурный сегмент, сегмент данных и динамический сегмент памяти (если используется страничная организация памяти, то указатели на таблицы страниц, соответствующих этим сегментам).

2.5 Создание процесса в ОС UNIX

+Состояние системы

В каждый момент времени ЦП может находиться в одном из трех состояний:

- **Система** — если выполняется команда ядра ОС.
- **Процесс** — если выполняется команда пользовательского процесса (процесс — концепция ресурсов, поток — концепция планирования).
- **Ожидание** — если в системе нет процессов, готовых к выполнению.

Переход «Система» -> «Процесс» осуществляется, когда запускается пользовательский процесс.

Переход «Процесс» -> «Система» может быть вызван одной из следующих причин:

- Прерывание по таймеру.
- Прерывание от устройств ввода-вывода.
- Обработка системного вызова.

Переход «Процесс» -> «Ожидание» когда нет потоков, готовых к выполнению.

Переход «Ожидание» -> «Процесс» при запуске программы.

Переход «Ожидание» -> «Система» НЕВОЗМОЖЕН!

Таймер регулярно с определенной частотой формирует прерывания, после которого управление автоматически передается специальной программе ядра, которая его обслуживает.

Для готовых к выполнению процессов активизируется процесс, имеющий наивысший приоритет.

Прерывание от устройств ввода-вывода возникает в момент завершения обмена информацией между ОП и периферийным устройством, т.е. когда устройство сообщает системе о завершении операции ввода-вывода.

Пользовательский процесс может обращаться к ядру ОС для выполнения системных вызовов.

Выполнение ядром системного вызова завершается или возвратом управления вызывающему процессу, или активизацией нового процесса.

Часто объем основной памяти не позволяет разместить в ней все имеющиеся процессы, часть располагается на внешней памяти в области свопинга.

Создание процесса в ОС UNIX

Все процессы в ОС UNIX, за исключением процесса с PID = 0 (swapper), создаются по одной и той же схеме при помощи СВ **fork()**.

Например, **./a.exe** — запуск **fork()** или **./a.exe &** — запуск **fork()**.

Этапы создания процесса:

1. Запуск СВ **fork():**

- a) Создание новой записи в таблице дескрипторов процессов ядра ОС UNIX.
- b) Копирование атрибутов процесса-родителя в дескриптор процесса-потомка.
- c) Присвоение идентификатора новому процессу.

3. Формирование контекста процесса:

- a) Копирование машинных регистров.
- b) Копирование таблицы пользовательских дескрипторов файлов, открытых процессом-родителем.
- c) Формирование оставшихся полей дескриптора.

4. Настройка карты памяти нового процесса:

- a) Выделение новых таблиц страниц, указывающих на таблицы родительского процесса. Эти таблицы страниц доступны только для чтения.
- b) Выделение новых страниц при попытке записи.

Рассмотрим подробнее этапы 2-4:

Формирование дескриптора нового процесса.

Когда выполняется системный вызов **fork()**,зывающий процесс обращается в ядро ОС и ищет свободную строку в таблице процессов, в которую можно записать данные о дочернем процессе. Если свободная строка находится, **fork()** копирует в нее все атрибуты родительского процесса. Затем присваивает новому процессу имя, т.е. меняет один из атрибутов — идентификатор процесса (дочерний процесс получает свой идентификатор).

Формирование контекста процесса.

Формирование контекста процесса связано с тем, что необходимо переписать машинные регистры родительского процесса в машинные регистры процесса-потомка, т.е. создать чистую копию родительского процесса. СВ **fork()** выделяет память для сегмента данных и динамического сегмента, куда копируются соответствующие сегменты родительского процесса. Контекст копируется вместе с динамическим сегментом. Процедурный сегмент может либо копироваться, либо использоваться совместно (если он доступен только для чтения). Далее, потомку предоставляется доступ к открытым файлам родительского процесса, посредством копирования дескрипторов файлов (ТПДОФ процесса) из контекста родительского процесса.

После формирования машинных регистров и ТПДОФ процесса, формируются оставшиеся три поля контекста процесса: информацию о состоянии текущего системного вызова, учетную информацию (пока что нулевая, т.к. процесс-сын еще никак не использовал процессорное время), стек ядра (также нулевой).

Настройка карты памяти нового процесса.

Карта памяти процесса — это тоже запись в таблице, которую ведет ОС (какие блоки или страницы памяти занимает процесс в ОП). Карта памяти процесса-сына настраивается по следующему принципу: сыну выделяются новые таблицы страниц, но эти таблицы указывают на страницы отца, помеченные, как доступные только для чтения. Когда процесс-сын пытается что-то записать в такую страницу, происходит страницное прерывание; при этом ядро ОС выделяет процессу-сыну новую копию этой страницы, к которой сын получает доступ на запись. Копируются только те страницы, в которые сын пишет новые данные; при этом страницы с процедурным сегментом не копируются.

Процесс-сын создан. При успешном завершении системного вызова **fork()**, процессы потомка и предка равноправно существуют в системе. Они могут функционировать параллельно, конкурируя за ресурсы на

основе своих приоритетов, или выполнение предка может быть приостановлено до завершения потомка системным вызовом `wait()`. Обычно процесс-сын создается для того, чтобы выполнить для родителя какие-то действия, т.к. с т.з. распределения ресурсов процесс-отец и процесс-сын являются конкурентами — у них общие ресурсы и они замедляют действия друг друга. Поэтому чем быстрее процесс-сын завершит свое существование, тем выгоднее для процесса-родителя.

2.6 Планирование процессов в ОС UNIX

Определение последовательности выполнения потоков предусматривает два алгоритма, которые работают независимо друг от друга. Оба алгоритма реализованы в диспетчерском процессе с PID = 0 (swapper):

1. **Низкоуровневый алгоритм** — выбирает следующий процесс из набора процессов в памяти, готовых к работе.
2. **Высокоуровневый алгоритм** — перемещает процессы из памяти на диск и обратно, т.е. осуществляет свопинг. Говоря иначе, он осуществляет загрузку в ОП образов активных процессов и выгрузку в область свопинга образов пассивных процессов.

Низкоуровневый алгоритм

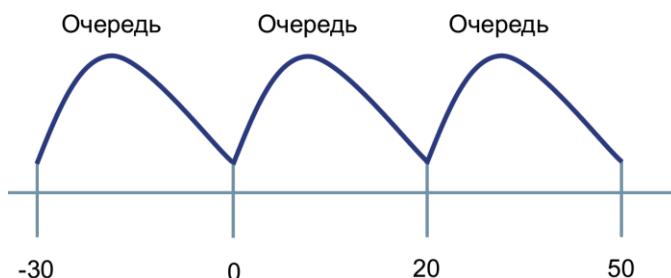
Задача низкоуровневого алгоритма — сформировать очередь процессов на выполнение.

1. Формирование очереди процессов на выполнение

В ОС UNIX предусмотрены несколько очередей, с каждой очередью связан диапазон непересекающихся значений приоритетов. В низкоуровневом алгоритме используется несколько очередей, обычно в ОС семейства UNIX 3-4 очереди.

- Процессы, выполняющиеся в режиме «процесс» обычно имеют положительное значение приоритетов.
- У процессов, выполняющих в режиме системы, значение приоритетов, как правило, отрицательное. Отрицательное значение приоритетов считается наивысшим. Сама приоритетная очередь имеет приоритеты от -30 до 0.

Когда запускается низкоуровневый алгоритм, он ищет очередь, начиная с самого высокого приоритета, т.е. наименьшего отрицательного значения, пока не находит очередь, в которой есть хотя бы один процесс. После этого в этой очереди выбирается и запускается первый процесс. Если процесс использует весь свой квант времени (~100 мс), он помещается в конец очереди, а алгоритм планирования запускается снова.



2. Пересчет приоритета процесса

Очереди постоянно пересматриваются ОС за счет низкоуровневого алгоритма и, если процесс находился в третий раз в первой очереди с высоким приоритетом, то через некоторое время он окажется во второй очереди, а в последствии может перейти даже в третью очередь. **Один раз в секунду (администратор может изменить этот интервал) приоритет каждого процесса пересчитывается по следующей формуле:**

$$PRI = \mu \cdot CPU + \eta \cdot NICE + k \cdot BASE$$

На основе нового приоритета каждый процесс прикрепляется к соответствующей очереди.

Здесь:

PRI — общий приоритет процесса, фиксируется в дескрипторе процесса. Приоритет — целое число. Чем меньше это число, тем выше приоритет у процесса. Если число отрицательное, то процесс с очень высоким приоритетом.

CPU — среднее значение тиков таймера в течении которых процесс выполнялся. При каждом тике таймера счетчик использования ЦП процессом в таблице процессов увеличивается на 1. Важно отметить, что при расчете приоритета процесса берется усредненное значение и за счет коэффициента оно отличается от значения в дескрипторе процесса, где приводится общее число тиков таймера, которое произошло с момента начала работы процесса (счетчик времени, в течении которого работал процесс).

NICE — Пользовательская составляющая приоритета процесса, диапазон значений $-20 \leq \text{NICE} \leq 20$.

Изменяется с помощью СВ **nice()** в диапазоне от 0 до 20. Системный администратор (привилегированный пользователь) может запросить обслуживание с более высоким приоритетом от -20 до 20. Это увеличивает частоту эксплуатации ЦП этим процессом.

BASE — характеристика блокированного процесса. Характеризует отрицательным числом, которое добавляется к приоритету процесса при выходе его из состояния «блокирован». Выбор значения (а, соответственно, и очереди) определяется событием, которого ждал процесс. Отрицательное значение приоритета BASE для дискового ввода-вывода (-2), терминального ввода-вывода (-4), выполнения различных системных вызовов (-1) и ожидания событий, жестко заданы в ОС, устанавливаются при инициализации системы и могут быть изменены только при новой инсталляции системы.

Когда процесс эмулирует прерывание для выполнения системного вызова, он блокируется, пока системный вызов не будет выполнен и процесс не вернется в режим «процесс». Блокировка происходит и при ожидании какого-либо события (**wait**) и при вводе-выводе. Когда процесс блокируется, он удаляется из очереди на выполнение. Однако, когда происходит разблокировка процесса (наступает событие, которого он ждал, или завершена операция ввода-вывода и т.п.), процесс снова помещается в очередь с отрицательным значением составляющей приоритета BASE.

μ, η, k — коэффициент для каждой из трех составляющих приоритета. Для каждой конкретной системы UNIX эти коэффициенты разные. Иногда это число, иногда функция вида $\mu(t)$.

Высокоуровневый алгоритм

Перемещает процессы из ОП на диск (во внешнюю память) и обратно (т.е. осуществляется свопинг / swapping). Отвечает за распределение и перераспределение оперативной памяти.

Т.е. если ОП не хватает для всех процессов, часть процессов приходится хранить во внешней памяти и за это перемещение также отвечает диспетчерский процесс с PID = 0 (swapper).

3. Системные вызовы, функции и утилиты управления процессами в ОС UNIX

Обработка ошибок

- Лав Р. *Linux. Системное программирование.* 2-е изд. — СПб.: Питер, 2014 г., стр. 50-53.
- Стивен Р.У., Раго С.А. *UNIX. Профессиональное программирование.* 3-е издание. — СПб.: Питер, 2018 г., стр. 48-50.
- Робачевский А., Немнюгин С., Стесик О. *Операционная система UNIX.* 2-е изд. — СПб.: БХВ-Петербург, 2010 г., стр. 109-115.
- Хэвиленд К., Грэй Д., Салама Б. *Системное программирование в UNIX: Руководство программиста по разработке ПО* — М.: ДМК Пресс, 2000 г., стр. 51-58.
- Керрик М. *Linux API. Исчерпывающее руководство.* — СПб.: Питер, 2019 г., стр. 84-86.

3.1 Процессы, управляемые терминалами, и фоновые процессы (демоны)

Процесс управляемый терминалом (запускается из терминала, использует ввод и вывод на терминал) создается после обычного запуска программы:

./lab

— запуск исполняемого файла lab, который находится в текущем каталоге. ОС формирует процесс и она не освободит терминал до тех пор, пока процесс не завершит свое выполнение.

Фоновый процесс (процесс демон) — процесс, не управляемый терминалом (не привязанный к терминалу). Он не может (не должен) читать из терминала (клавиатура) и писать на терминал (монитор).

Создается после запуска программы:

./lab&

— запуск исполняемого файла lab в фоновом режиме (добавление знака & говорит, чтобы после запуска этой программы ОС освободила терминал). В ходе выполнения этого фонового процесса можно запускать с терминала новые инструкции.

Просмотр существующих в ОС процессов

Утилита **ps** (process status) позволяет отслеживать процессы, которые существуют в настоящее время в ОС.

Вывод на монитор информации обо всех процессах, ассоциированных с данным терминалом пользователя (все процессы, запущенные с данного терминала):

ps

Вывод на монитор информации обо всех процессах ОС управляемых терминалами:

ps -a

Вывод на монитор информации обо всех фоновых процессах ОС:

ps -x

Вывод на монитор информации обо всех существующих активных процессах ОС (ключ I — вывод полной информации о процессах):

ps -axl

3.2 Информационные связи между параллельными процессами

Информационная связь между процессом и ядром отсутствует, за исключением передачи внешних аргументов.

Информационная связь между пользовательскими процессами осуществляется либо по одноранговой схеме, либо по многоранговой (клиент-сервер).

Одноранговая схема (связанная с пространством ОП пользователя) предусматривает следующие возможности обмена информацией между процессами:

1. Через **межпроцессный канал (именованный или неименованный)**. Межпроцессный канал — особый файл с рабочей диспозицией.
2. Через **файловую систему** (один процесс пишет в файл, другой из этого файла читает).
3. Через **область внешних аргументов**.
4. (Не рассказывать — найдено вне лекции) Через **переменные среды или переменные окружения (environment variables)** — текстовые переменные операционной системы, хранящие какую-либо информацию — например, данные о настройках системы. Подробнее см. в статье [Как читать и определять переменные окружения и оболочки на Linux](#).

Переменные среды устанавливаются пользователем или сценариями оболочки. Важной особенностью является то, что при создании дочерний процесс получает локальную копию среды родительского процесса, а, значит, не может изменить напрямую глобальные установки. Когда создается новый процесс, он наследует копию среды своего родителя. Это простая, но часто используемая форма межпроцессного взаимодействия (IPC) — среда предоставляет способ переноса информации от родительского процесса его дочернему процессу или процессам. Поскольку дочерний процесс при создании получает копию среды своего родительского процесса, этот перенос информации является односторонним и однократным. После создания дочернего процесса любой процесс может изменить свою собственную среду, и эти изменения будут незаметны для других процессов. Среда представляет собой набор пар «имя переменной» и «значение переменной», реализация которыми возложена обычно на командный интерпретатор. Как «имя», так и «значение» чувствительны к регистру символов, «Имя» обычно указывается в верхнем регистре, использование пробелов недопустимо.

Межпроцессное взаимодействие по **многоранговой схеме (клиент-сервер)**, связанной с **пространством**

ОП ядра ОС осуществляется следующими способами:

1. При помощи **очереди (канала) сообщений** — позволяет процессам, работающим на одном компьютере, обмениваться форматированными данными.
2. При помощи **разделяемой памяти и семафоров**. Этот способ позволяет нескольким процессам совместно использовать общую область памяти. **Семафоры** — общесистемные переменные, находящиеся в ядре ОС (эти переменные видят все процессы), которые могут модифицироваться и использоваться процессами, запущенными на одном компьютере, для синхронизации их выполнения. Семафоры обычно используются в сочетании с разделяемой памятью для управления доступом к данным, находящимся в той или иной области разделяемой памяти.
3. Через **сокеты (гнезда)**. Интерфейс транспортного уровня (sockets) — позволяет процессам, выполняемым на разных компьютерах, создать прямой двусторонний канал связи.

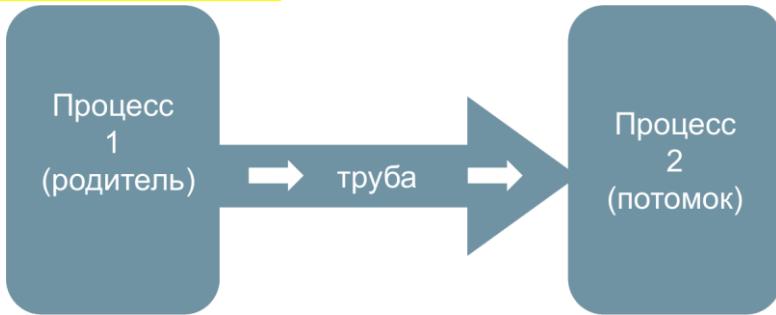
Обмен информацией через межпроцессный канал (программный, коммуникационный, неименованный, байтовый)

Создается (неименованный канал) при помощи СВ **pipe()**. Используется для связи между родственными процессами. В ОП создает файл, предназначенный для создания/открытия временного буфера (канала), обеспечивающего взаимодействие двух или нескольких процессов посредством записи данных в этот буфер и чтения данных из буфера. **Межпроцессный канал** — особый файл с рабочей диспозицией, создаваемый в ОП в пространстве пользователя. Это однонаправленные файлы (полудуплексные), работающие по схеме FIFO «first-in-first-out» (первый пришел — первый ушел).

Между асинхронными процессами такой (неименованный) канал организовать нельзя, поэтому обычно создается процесс-сын. Когда создается межпроцессный канал, получаются два пользовательских дескриптора открытого файла в одном процессе, затем посредством СВ **fork()** создается дочерний процесс, у которого также будет два пользовательских дескриптора этого файла. Затем дескрипторы открытого файла настраиваются таким образом, чтобы направление канала было в одну сторону. После этого один процесс

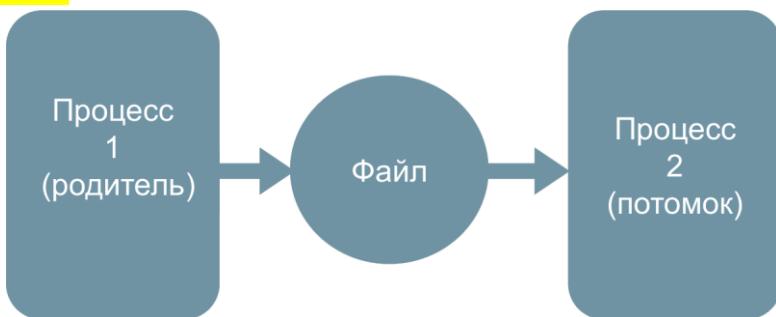
записывает в канал, а другой из него читает. За синхронизацию действий записи и чтения из канала следит сама ОС.

Буфер (файл) удаляется, когда все процессы, связанные с FIFO-файлом, закрывают все свои ссылки на него (межпроцессный канал).



Обмен информацией через файловую систему

Один процесс пишет в файл, другой из этого файла читает. Когда осуществляется обмен информацией через файл, нужно синхронизировать действия процессов, т.е. пока один процесс пишет в файл, другой процесс не должен читать информацию из файла. В отличии от межпроцессного канала, этот способ обмена выполняется медленнее, т.к. файл обычно находится во внешней памяти, которая работает медленнее, чем ОП.



Обмен информацией через область внешних аргументов

В рамках одного процесса формируются байтовые (или символьные, если передается текст) массивы, куда записывается информация, которую нужно передать в другой процесс. Затем в рамках данного процесса запускается другой процесс с помощью СВ из группы `exec()` с внешними аргументами, переданными в виде массива(ов). Во вновь порожденном процессе можно посмотреть значения этих внешних аргументов и обработать их.

Формально аргументами называются все слова в командной строке (в том числе и имя самой команды), разбитые разделителем (как правило, это пробел и табуляция), кавычки же позволяют включать разделители в аргументы.

Как в коде программы можно прочитать ее внешние аргументы? Функция `main()` в C определяется так:

```

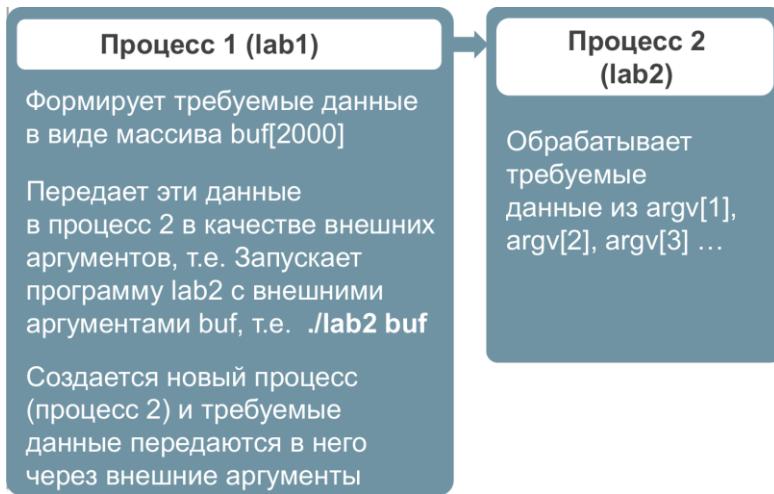
int main(int argc, char *argv[])
  
```

Здесь присутствует два параметра: `argc` определяет количество аргументов в командной строке, а `argv` хранит массив указателей на эти аргументы.

Следует отметить, что `argv[0]` — всегда имя команды, а `argv[argc] == NULL`, эти два факта могут оказаться полезными при разработке.

Подробнее про аргументы командной строки (`argc, argv`) и функции разбора аргументов командной строки в UNIX-подобных ОС можно прочитать [здесь](#), [здесь](#) и в книге Керриск M. Linux API. Исчерпывающее руководство. — СПб.: Питер, 2019 г. стр. 156-158.

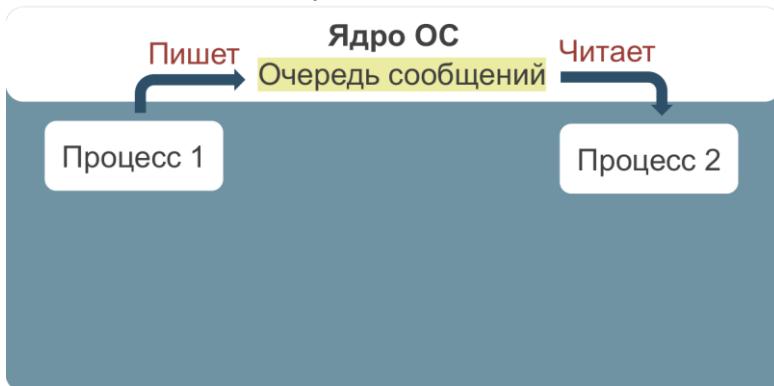
В UNIX-подобных ОС имеются ограничения на максимальную длину аргументов вызовов семейства `exec` (см. Стивен Р.У., Раго С.А. UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018 г., стр. 77-81, 90, 311-312)



Обмен информацией через очередь сообщений (канал сообщений)

Очередь (канал) сообщений позволяет процессам, работающим на одном компьютере, обмениваться форматированными данными. Создается одним процессом, другой процесс может открыть эту очередь сообщений и читать/писать в эту очередь сообщений. Обмен информацией через очередь сообщений синхронизирован самой ОС, не надо задумываться, как синхронизировать этот обмен. Очередь сообщений создается в пространстве памяти ядра ОС — это буфер, в который можно писать и из которого можно считывать информацию. Также, как и межпроцессный канал, представляет собой файл, работающие по схеме FIFO «first-in-first-out» (первый пришел — первый ушел) и при создании или открытии канала сообщений процесс получает пользовательский дескриптор открытого файла. Однако, в отличии от межпроцессного канала, канал сообщений создается в ОП в пространстве памяти ядра ОС. Другое отличие состоит в том, что межпроцессный канал и именованный каналы — байтовые (3 входящих «сообщения» на выходе сливаются в одно большое сообщение), а канал сообщений — не байтовый (3 входящих «сообщения» на выходе вернут 3 сообщения, причем сообщения из канала можно извлекать выборочно). Когда сообщение считывается из канала сообщений хотя бы одним процессом, его уже там не будет.

Оперативная память



Обмен информацией через разделяемую память

Этот способ позволяет нескольким процессам совместно использовать общую область памяти.

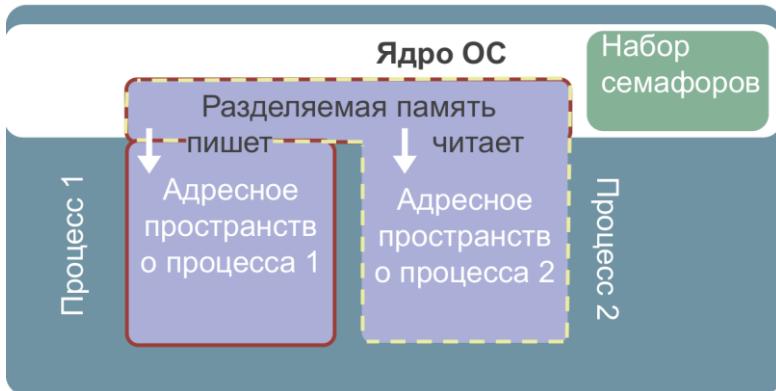
В ОП ядра ОС один из процессов создает блок памяти. Обычно следует выделять объем память размером 4 кБ, т.к. UNIX-подобные ОС разбивают ОП на страницы или одинаковые блоки такого размера. Если выбрать меньший размер, то ОС округлит объем разделяемой памяти до размера страницы (4 кБ). После создания такой памяти процесс может подсоединить эту память к своему адресному пространству и любой из других параллельно работающих процессов также может подсоединить эту память к своему адресному пространству, тем самым увеличивая адресное пространство другого процесса. Эта память может оказаться общей у нескольких процессов, любой из которых может как читать, так и писать в эту память.

Оперативная память



Проблема синхронизации обращения параллельных процессов к этому участку памяти обычно решается с помощью семафоров. **Семафоры** — общесистемные переменные, находящиеся в ядре ОС (эти переменные видят все процессы), которые могут модифицироваться и использоваться процессами, запущенными на одном компьютере, для синхронизации их выполнения. Семафоры обычно используются в сочетании с разделяемой памятью для управления доступом к данным, находящимся в той или иной области разделяемой памяти.

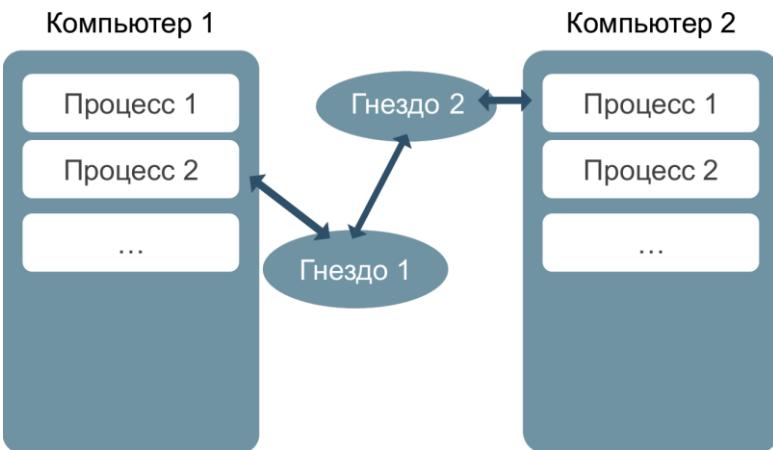
Оперативная память



Обмен информацией через гнезда

Через сокеты (гнезда). Интерфейс транспортного уровня (**sockets**) позволяет процессам, выполняемым на разных компьютерах, создать прямой двусторонний канал связи. Гнезда можно использовать и для обмена информации между процессами, работающими на одном компьютере, но обычно этого не делают, т.к. из-за накладных расходов обмен будет осуществляться значительно медленнее. Через аппарат гнезд построена вся сетевая поддержка в ОС (выход в интернет, взаимодействие между любыми двумя компьютерами и т.д.).

Процесс, работающий на одном компьютере, создает гнездо. Процесс, работающий на другом компьютере, также создает гнездо. **Гнезда** — это файлы, которые используются для обмена информации по сети. У каждого такого файла есть свое имя, в качестве имени гнезда используется сетевой адрес компьютера. Это может быть как полное имя файла в системе UNIX, так и IP-адрес компьютера и номер порта. Весь обмен информацией осуществляется именно через эти файлы. Один процесс, работающий на одном компьютере, пишет в свой файл (гнездо), а другой процесс, который работает на другом компьютере, читает информацию из своего файла (гнезда). При этом информация проходит по каналу связи (выделенному, радиоканалу и т.п.), который можно проверить с помощью команды **ping**. Гнезда поддерживают протоколы TCP, IP, UDP (для гнезд дейтаграммного типа).



3.3 Запуск новой программы в рамках выполняющейся

Системный вызов system

Простейшим способом вызвать другую программу / консольную команду (сформированную в виде командной строки) из уже выполняющейся является СВ **system()**:

```
#include <stdlib.h>
int system(const char* cmd);
```

cmd — указатель на массив, содержащий утилиту командной строки, включая любые дополнительные аргументы. Позволяет в рамках выполняющейся программы (процесса) вызвать новую программу (процесс).

СВ **system()** запускает команду, предоставляемую параметром **cmd**, включая любые дополнительные аргументы. Символьный массив **cmd** используется для формирования команды, включающей имя программы и множество ее внешних аргументов (если они присутствуют), разделенных между собой пробелами.

В случае успеха возвращает **0**, в случае неудачи **-1**.

СВ **system()** удобно использовать для того, чтобы не выходить из контекста данного процесса. Другие СВ семейства **exec()**, использующиеся для запуска новых программ в рамках уже существующих программ, не позволяют сохранить контекст текущего процесса.

Выдержка из книги Лав Р. Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014 г., стр. 194-195:

Функция **system()** называется так потому, что синхронный запуск процессов называется выходом в систему. Принято использовать **system()** для запуска простой утилиты или сценария оболочки, когда основной целью является получение возвращаемого ими значения.

Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX: Руководство программиста по разработке ПО — М.: ДМК Пресс, 2000 г., стр. 313-314:

Вначале СВ **system()** создает дочерний процесс, который, в свою очередь, осуществляет вызов **exec()** для запуска стандартного командного интерпретатора UNIX с командной строкой **cmd**. В это время **system()** в первом процессе выполняет СВ **wait()**, гарантируя тем самым, что выполнение продолжится только после того, как запущенная команда завершится. Возвращаемое после этого значение содержит статус выхода командного интерпретатора, по которому можно определить, было ли выполнение программы успешным или нет. В случае неудачи любого из вызовов **fork()** или **exec()** возвращаемое значение будет равно **-1**. Поскольку в качестве посредника выступает командный интерпретатор, строка **cmd** может содержать любую команду, которую можно набрать на терминале. Это позволяет программисту воспользоваться такими преимуществами командного интерпретатора, как перенаправление ввода-вывода, поиск файлов в пути и т.д.

Керрик M. Linux API. Исчерпывающее руководство. — СПб.: Питер, 2019 г., стр. 590-591:

Принципиальными преимуществами функции **system()** являются простота и удобство:

- Нам не нужно иметь дела с деталями вызовов **fork()**, **exec()**, **wait()** и **exit()**.
- Обработка ошибок и сигналов выполняется за нас самой функцией **system()**.

- Поскольку **system()** использует для выполнения команды в командной оболочке, перед ее запуском выполняются все стандартные процедуры обработки, подстановки и перенаправления. Это упрощает добавление в приложение возможности вида «выполнить консольную команду» (подобная функция доступна во многих интерактивных программах по команде !).

Главным недостатком функции **system() является ее низкая производительность.** Запуск команды с ее помощью требует создания как минимум двух процессов — одного для командной оболочки, и еще одного — для выполняемой команды; каждый такой процесс требует вызова **exec()**. Если эффективность или скорость являются важным требованием, для запуска нужной программы лучше воспользоваться непосредственно вызовами **fork()** и **exec()**.

Примеры

```
#include <stdlib.h>
/* Код программы */
...
system("ls"); // утилита ls выводит на экран список файлов текущего каталога
              // (имена, начинающиеся с точки, не выводятся)
... // текущий процесс продолжает свое выполнение
// копирование файла a.txt в файл b.txt:
system("cp a.txt b.txt"); // argc = 3 команда с тремя внешними аргументами
... // argv[0] = "cp"
... // argv[1] = "a.txt"
... // argv[2] = "b.txt"
... // текущий процесс продолжает свое выполнение
system("./lab.exe"); // запуск исполняемого файла lab.exe из текущей директории
... // текущий процесс продолжает свое выполнение
```

Системные вызовы семейства exec()

Вызывают новую программу, вместо уже выполняющейся без возврата в вызывающую программу. Иными словами, **позволяют запустить новую программу в рамках уже выполняющейся и при этом сменить контекст текущего процесса на контекст нового процесса.** Т.е. корректный запуск СВ из семейства **exec()** однозначно приведет к тому, что текущий процесс будет завершен и в рамках этого процесса будет создан новый процесс, который продолжит свою работу.

«В рамках этого процесса будет создан новый процесс» означает, что **идентификатор процесса не изменяется после вызова СВ семейства exec()**. Кроме того, новая программа наследует от вызывающего процесса ряд дополнительных характеристик, описываемых в контексте и (отчасти) в дескрипторе процесса.

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char* const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Все семь функций возвращают -1 в случае неудачи, в случае успеха управление не возвращает.

СВ **execl() полезен только для вызова программ с фиксированным числом аргументов (или, по крайней мере, когда существует небольшая фиксированная верхняя граница числа аргументов).** Рассмотрим **синтаксис СВ **execl()**, внешние аргументы в который передаются списком переменной длины:**

```
int execl(const char* path, const char* arg0, const char* arg1, ..., NULL);
```

*path — указатель на строку символов, содержащую полное имя файла (указываются все подкаталоги, начиная с корневого) вызываемой программы. Большинство утилит находится в каталогах /bin и /usr/bin.

*arg0, *arg1, *arg2, ... — внешние аргументы вызываемой программы, где *arg0 — имя программы, *arg1 — первый внешний аргумент и т.д.— для СВ, заканчивающихся на I.

Конец списка аргументов отмечается пустым указателем NULL, чтобы СВ мог определить, какой именно аргумент является последним.

Вызывает на выполнение указанную программу, заменяя контекст текущего процесса на контекст нового процесса, т.е. по функции exit(0)/exit(1) завершает текущий процесс и создает новый процесс для указанной программы.

Чаще всего СВ execv() — лучший выбор, поскольку он позволяет вызвать программу с произвольным числом аргументов, когда аргументы определяются во время выполнения программы. Рассмотрим синтаксис СВ execvp(), внешние аргументы в который передаются массивом указателей:

```
int execvp(const char *file, char *const argv[]);
```

*file — неполное имя файла вызываемой программы (файл будет искааться в текущем каталоге и в пределах пользовательского пути, т.е. в каталогах, перечисленных в переменной окружения PATH) — для СВ, заканчивающихся на p.

*argv[] — массив указателей char *argv[] на строки (символьные массивы) с аргументами командной строки (по одному указателю на каждый аргумент и один завершающий). Должно присутствовать как минимум 2 аргумента: имя программы, которое отобразится в элемент argv[0], и значение NULL, завершающее список. Для СВ, заканчивающихся на v, последним элементом массива должен быть нулевой указатель, чтобы СВ смог определить, какой именно аргумент является последним.

Нужно упомянуть, что в UNIX-подобных ОС имеются ограничения на максимальную длину аргументов вызовов семейства exec (см. [здесь](#), [здесь](#), а также Стивен Р.У., Раго С.А. UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018 г. стр. 77-81, 90, 311-312 и Керриск М. Linux API. Исчерпывающее руководство. — СПб.: Питер, 2019 г., стр. 156-158).

О различиях между СВ группы exec ()

Главные различия между СВ группы exec() состоят в количестве внешних аргументов и в разных форматах внешних аргументов (они передаются списком или массивом указателей).

Выдержка из книги Лав Р. Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014 г., стр. 175-178:

Единой функции exec не существует; на одном системном вызове построено целое семейство таких функций. Запомнить все очень просто. Символы I и v в имени указывают, передаются ли аргументы списком переменной длины (list) или массивом указателей (vector). Символ p указывает, что система будет искать указанный файл по полному пользовательскому пути. В командах, где используются варианты с буквой p, можно указать только имя файла, если он находится в пределах пользовательского пути (используется переменная окружения PATH, чтобы найти выполняемый файл). Наконец, e обозначает, что для нового процесса создается новое окружение. Интересно, что, хотя технических ограничений для этого не существует, в семействе exec нет элемента, позволяющего искать путь к файлам, и создавать новое окружение. Возможно, это объясняется тем, что варианты p предназначены для использования оболочками, а процессы, исполняемые в оболочках, как правило, наследуют свое окружение от них.

Элементы семейства exec , принимающие в качестве аргумента массив, работают точно так же, за одним исключением — вместо списка генерируется и передается массив. Использование массива позволяет определять аргументы во время выполнения программы. Как и в случае аргумента в виде списка переменной длины, массив должен заканчиваться значением NULL.

Пример 1

```
#include <unistd.h>
/* Код программы */
...
int k;
k = execl("/bin/ls", "ls", NULL); /* После выполнения этого СВ текущий процесс
                                     прекратит свое существование и выведется
                                     список файлов в текущей директории */
// arg0 = "ls" - единственный внешний аргумент, это имя программы
if(k== -1) /* произошла ошибка, в случае которой текущий процесс
             продолжает свое выполнение */
```

Пример 2

```
#include <unistd.h>
/* Код программы */
...
int k;
k = execl("/bin/cp", "cp", "a.txt", "b.txt", NULL); /* После выполнения этого СВ
                                                       текущий процесс прекратит
                                                       свое существование, файл
                                                       a.txt будет скопирован
                                                       в файл b.txt */
// Команда с тремя внешними аргументами:
// arg0 = "cp"
// arg1 = "a.txt"
// arg2 = "b.txt"
if(k== -1) /* произошла ошибка, в случае которой текущий процесс
             продолжает свое выполнение */
```

Пример 3

```
#include <unistd.h>
/* Код программы */
...
int k;
// запуск исполняемого файла lab.exe из текущей директории
k = execl("./lab.exe", "lab.exe", NULL);
// arg0 = "lab.exe" - единственный внешний аргумент, это имя программы
if(k== -1) /* произошла ошибка, в случае которой текущий процесс
             продолжает свое выполнение */
```

Отличия СВ `execl()` от `system()`

СВ `execl()` завершает процесс, в рамках которого он запущен (и подменяет этот процесс новым процессом), а СВ `system()` — нет. Поэтому принято использовать СВ `system()` для запуска простой утилиты или сценария оболочки, когда основной целью является получение возвращаемого ими значения. СВ `system()` вызывает командную оболочку системы (`sh` по умолчанию), которая выполнит командную строку, переданную в качестве аргумента, т.е. как минимум будет создан процесс командной оболочки, а также, возможно, могут быть созданы дополнительные процессы (в зависимости от вызываемой команды и системы). СВ `system()` удобно использовать для того, чтобы не выходить из контекста данного процесса. Другие СВ семейства `exec()`, использующиеся для запуска новых программ в рамках уже существующих программ, не позволяют сохранить контекст текущего процесса. Т.е. при использовании СВ семейства `exec()` исходная программа больше не будет работать после запуска новой.

СВ `execl()` позволяет передавать информацию через внешние аргументы в параллельный процесс, а `system()` — нет. Точнее говоря, СВ `system()` запускает команду, предоставленную символьным массивом, включающим как имя команды, так и множество ее внешних аргументов, разделенных между собой пробелами. Но передать отдельно имя программы от ее внешних аргументов нельзя.

Керрик M. *Linux API. Исчерпывающее руководство*. — СПб.: Питер, 2019 г., стр. 590-591:

Главным недостатком функции `system()` является ее низкая производительность. Запуск команды с ее помощью требует создания как минимум двух процессов — одного для командной оболочки, и еще одного — для выполняемой команды; каждый такой процесс требует вызова `exec()`. Если эффективность или скорость являются важным требованием, для запуска нужной программы лучше воспользоваться непосредственно вызовами `fork()` и `exec()`.

3.4 Создание нового процесса

Системные вызовы семейства `fork()`

Для того, чтобы запустить параллельно новую программу, необходимо запускать параллельные процессы. Для этого применяется СВ `fork()`, который позволяет получить дубликат текущего процесса. После его применения процесс разделяется на две идентичные копии, которые продолжают выполняться как два независимых процесса. СВ `fork()` можно использовать в рамках любой программы:

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork(void);
pid_t fork(void);
```

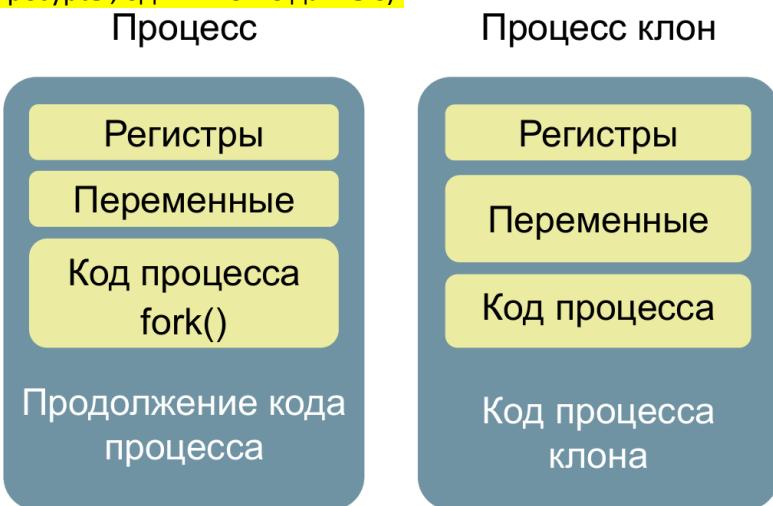
Возвращаемые значения: процесс-потомок получает от `fork()` код ответа **0**, в то время как процесс-родитель — идентификатор, под которым запущен процесс-потомок. В случае аварийного завершения СВ `fork()` вернет **-1**.

```
if(fork() == 0)
{ /* Код процесса-потомка*/ }
else
{ /* Код процесса-родителя*/ }
```

После выполнения СВ `fork()` программа делится на две абсолютно идентичные копии (те же самые переменные, те же самые регистры, те же самые открытые файлы, те же самые ресурсы), создается новый процесс. Как уже было сказано выше, одна из копий получает **код ответа 0 (процесс-потомок)**, в то время, как другая копия (**процесс-отец**) **получает идентификатор процесса-сына**.

После создания каждый из процессов начинает «жить своей жизнью». Обычно СВ `fork()` используется для того, чтобы создать процесс-сын, который бы выполнил для процесса-отца какую-то работу и после выполнения этой работы процесс-отец обычно завершает процесс-сын, чтобы процесс-отец выполнялся

быстрее, поскольку процесс-сын является конкурентом процессу-отцу в рамках ОС (они используют одни и те же ресурсы, одни и те же данные).



Стивен Р.У., Раго С.А. *UNIX. Профессиональное программирование*. 3-е издание. — СПб.: Питер, 2018 г., стр. 308-309

Функция `fork` часто используется для создания нового процесса, который затем запускает другую программу с помощью одной из функций семейства `exec`. Когда процесс вызывает одну из функций `exec`, он полностью замещается другой программой, и эта новая программа начинает выполнение собственной функции `main`. Идентификатор процесса при этом не изменяется, поскольку функция `exec` не создает новый процесс, она просто замещает текущий процесс — его сегмент кода, сегмент данных, динамическую область памяти и сегмент стека — другой программой.

Пример 1

```

#include <sys/types.h>
#include <unistd.h>
void main()
{
    int p;
    /* Код программы */
    p = fork();
    if (p != 0)
    {
        if (p < 0) { /* аварийное завершение СВ fork() */ }
        else { /* Код процесса-родителя*/ ... }
    }
    else { /* Код процесса-потомка*/ ... }
}
    
```

Аварийное завершение СВ `fork()` может случиться в случае, когда создается очень большое число процессов, превышающих максимальное возможное количество одновременно выполняемых параллельных процессов. В этом случае СВ `fork()` вернет `-1`. Такое случается очень редко и обычно пользователь пишет свою программу, не описывая аварийное завершение СВ `fork()`, как в примере 2.

Пример 2

```

#include <sys/types.h>
#include <unistd.h>
void main()
{ /* Код программы */
...
}
    
```

```

if (fork() == 0 )
{ /* Код процесса-потомка */ ... }
else { /* Код процесса-родителя*/ ... }
}

```

Завершение процесса

У ОС есть два способа завершить процесс: с помощью СВ **_exit()** или с помощью сигналов (сигнал — **программная версия аппаратных прерываний**).

Программа, в которой СВ **_exit()** не записан, завершается сама, потому что ОС, обратившись к этому процессу после того, как все команды закончились, обнаружит, что счетчик команд пуст, а процесс уже никак не откликается. Тогда ОС сама завершает данный процесс по СВ **_exit()**.

Нормальное завершение программы осуществляется тремя функциями: **_exit** и **_Exit**, сразу же возвращающими управление ядру, и **exit()**, выполняющей ряд дополнительных операций таких, как принудительный сброс всех буферов ввода-вывода, и только после этого возвращающей управление ядру. Все три функции закрывают все открытые в данном процессе файлы.

В большинстве версий UNIX **exit()** реализована как библиотечная функция, а **_exit()** — как системный вызов. Функция **exit()** является оберткой для СВ **_exit()**. В системе UNIX имена **_exit** и **_Exit** являются синонимами, обе функции не сбрасывают буферы ввода-вывода. Т.е. функция **printf()** ничего не сможет вывести и вернет признак ошибки, тогда как небуферизованный вывод через системный вызов **write()** будет выведен.

Синтаксис библиотечной функции exit():

```
#include <stdlib.h>
void exit(int status);
```

exit() — завершает процесс. Эта функция — надстройка над СВ **_exit()**. Выталкиваются все буферы ввода-вывода (выводит как **printf()**, так и **write()**), закрываются все открытые в данном процессе файлы.

status — аргумент, задаваемый пользователем, характеризует причину завершения процесса. Обычно значение аргумента задается равным 0, если процесс завершается удачно. Для кодов ошибок желательно использовать положительные числа. Отрицательные значения лучше не использовать, т.к. их сложнее классифицировать.

Синтаксис СВ _exit():

```
#include <unistd.h>
#include <stdlib.h>
void _exit(int status);
```

_exit() — немедленно завершает программу, закрываются все открытые в данном процессе файлы, буферы ввода-вывода не выталкиваются (выводит **write()**, тогда как **printf()** выведен не будет).

Примеры

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    printf("Hello World");
    exit(1);
}
```

```
ubuntu@ubuntu:~/dim/one$ ./a.out
Hello Worldubuntu@ubuntu:~/dim/one$ -
```

Завершение процесса путем вызова СВ `_exit()`, а не библиотечной функции `exit()` приводит к тому, что процесс завершается без сброса буферов stdio или вызова обработчиков выхода.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
    printf("Hello World");
    _exit(1);
}
ubuntu@ubuntu:~/dim/one/third$ ./a.out
ubuntu@ubuntu:~/dim/one/third$
```

3.6 Ожидание завершения «родственного» процесса

Системный вызов wait

СВ `wait()` используется в процессе-отце для того, чтобы подождать завершение процесса-сына. После системного вызова `fork()` процесс-родитель может посредством СВ `wait()` приостановить свое выполнение до завершения процесса-сына или продолжить свое выполнение независимо от процесса-сына.

В случае параллельной работы, с помощью СВ `signal()` процесс-отец может выявить или игнорировать завершение процесса-сына:

```
int signal(sigaction);
```

Синтаксис СВ `wait()`:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *s);
```

СВ `wait()` в случае ошибки возвращает **-1**, в случае успеха возвращает **идентификатор завершившегося процесса-сына** (если создано несколько сыновей, то это позволяет распознать, какой именно из сыновей завершился), а в свой единственный аргумент записывает **код завершения процесса-сына**. Как известно, у ОС есть два способа завершить процесс: с помощью СВ `_exit()` или с помощью сигналов (сигнал — программная версия аппаратных прерываний).



Код завершения можно представить в виде следующей битовой маски:

- 7 младших битов (с 0 по 6) содержат нули, если процесс-сын был завершен с помощью системного вызова `_exit()`, или номер сигнала, завершившего процесс.
- 8-й бит (бит номер 7) равен 1, если из-за прерывания процесса-сына был создан дамп образа процесса (core файл), т.е. можно ли потом продолжить приостановленный процесс-сын, или нет (он завершен).
- Если процесс был завершен с помощью системного вызова `_exit()`, то биты с 9 по 16-й (биты с номерами с 8 по 15-й) содержат аргумент системного вызова `_exit()`.

Как анализировать причину завершения процесса-сына? Если аргумент СВ `_exit()` равен нулю, то все биты также будут нулевыми (поскольку сигнала с нулевым номером нет и все сигналы начинаются с номера 1).

Если аргумент `CB_exit()` равен единице, то в 8-м бите будет единица и значение переменной $s = 2^8 = 256$ (см. пример ниже). Если аргумент `CB_exit()` равен тройке, то тройка в двоичной системе записывается как 11, поэтому в 8-м и 9-м битах будут единицы и значение переменной $s = 2^9 + 2^8 = 512 + 256 = 768$.

Стивен Р.У., Раго С.А. UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018 г., стр. 297-298:

Согласно стандарту POSIX.1, в файле `<sys/wait.h>` определяются различные макросы, с помощью которых производится извлечение кодов выхода. Определить, как завершился процесс, можно с помощью четырех взаимоисключающих макросов, имена которых начинаются с префикса `WIF`. В зависимости от того, какой из этих четырех макросов возвращает истину, можно использовать другие макросы, чтобы получать код выхода, номер сигнала и другую информацию.

Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX: Руководство программиста по разработке ПО — М.: ДМК Пресс, 2000 г., стр. 148-149:

Как уже упоминалось, сигналы *SIGABRT*, *SIGBUS*, *SIGSEGV*, *SIGQUIT*, *SIGILL*, *SIGTRAP*, *SIGSYS*, *SIGXCPU*, *SIGXFSZ* и *SIGFPE* приводят к аварийному завершению и обычно сопровождаются сбросом образа памяти на диск. Образ памяти процесса записывается в файл с именем *core* в текущем рабочем каталоге процесса (термин *core*, или сердечник, напоминает о временах, когда оперативная память состояла из матриц ферритовых сердечников). Файл *core* будет содержать значения всех переменных программы, регистров процессора и необходимую управляющую информацию ядра на момент завершения программы. Статус завершения процесса после аварийного завершения будет тем же, каким он был бы в случае нормального завершения из-за этого сигнала, только в нем будет дополнительно выставлен седьмой бит младшего байта.

Формат файла *core* известен отладчикам *UNIX*, и этот файл можно использовать для изучения состояния процесса в момент сброса образа памяти. Этим можно воспользоваться для определения точки, в которой возникает проблема.

Пример

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
void main()
{
    int s;
    /* Тело программы */
    if(fork() == 0)
    {
        /* Код процесса-потомка */
        _exit(1);
    }
    else
    {
        wait(&s);
        /* Код процесса-родителя*/
        printf("Причина гибели процесса-потомка %d\n", s);
    }
} // s = 256
```

Полезные системные вызовы для управления процессами

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
pid_t getpid(void);
```

getpid() — возвращает идентификатор вызывающего (текущего) процесса.

```
pid_t getppid(void);
```

getppid() — возвращает идентификатор родительского процесса.

```
uid_t getuid(void);
```

getuid() — получить фактический идентификатор (идентификатор пользователя запустившего процесс)

пользователя текущего процесса.

```
#include <unistd.h>
```

```
#include <sys/types.h>
```

```
gid_t getgid(void);
```

getgid() — получает фактический идентификатор группы вызывавшего процесса (про фактический выше).

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

kill() — посыпает сигнал одному или нескольким процессам. В т.ч. позволяет убить любой дочерний

процесс из родительского процесса.

```
#include <unistd.h>
```

```
int setpgid(pid_t pid, pid_t pgid);
```

setpgid() — устанавливает идентификатор группы процессов.

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
int setuid(uid_t uid);
```

setuid() — устанавливает фактический идентификатор владельца текущего процесса (позволяет передать

процесс другому владельцу).

4. Файлы и файловые системы в ОС UNIX

4.1 Понятия файла, файловой системы, дескриптора файла

Понятия файла

Для сохранения информации в компьютере используется такой абстрактный механизм как файл.

Файл — именованный набор данных, состоящий из записей, с заданной структурой, наложенной пользователем. Каждый файл на диске занимает какое-то количество блоков.

Файл состоит из набора взаимосвязанных записей.

Запись — набор взаимосвязанных полей.

Поле — набор взаимосвязанных байтов (символов — по 1, 2, или даже 4 байта на символ, в зависимости от выбранной кодировки).

Байт — набор взаимосвязанных бит (8 битов). В большинстве вычислительных архитектур байт — это минимальный независимо адресуемый набор данных.

Бит принимает значение 0 или 1 на машинном языке, это минимальная единица измерения информации, соответствующая одной двоичной цифре.

Поля разделяются символами: «пробел», «\t».

Записи разделяются символом: «\n».

Понятие символьного набора

Символьный набор — это кодировка символов. Любой символ можно представить в виде комбинации битов. Можно представлять символы с помощью одного байта ($2^8 = 256$ вариантов/символов), с помощью двух байт ($2^{16} = 65\,536$ вариантов/символов), с помощью четырех байт ($2^{32} = 4\,294\,967\,296$ вариантов/символов). Символьный набор кем-то предлагается и утверждается сообществом. Иногда он называется способом кодировки. Известные символьные наборы:

- ASCII (American standard code for information interchange) — американский стандарт кодировки, в основном используется для передачи данных по сети.
- UNICODE — стандарт кодирования символов почти всех письменных языков мира (UTF-8, UTF-16, UTF-32). UTF обозначает семейство кодировок Unicode transformation format.
- EBCDIC (Extended Binary Coded Decimal Interchange Code — расширенный двоично-десятичный код обмена информацией; произносится «Эб-си-дик») — стандартный восьмибитный код, разработанный корпорацией IBM для внутреннего использования на своих мейнфреймах. Российским аналогом EBCDIC является код ДКОИ-8 («двоичный код обработки информации»), в который добавлена кодировка кириллицы.

Файлы могут быть структурированы различными способами:

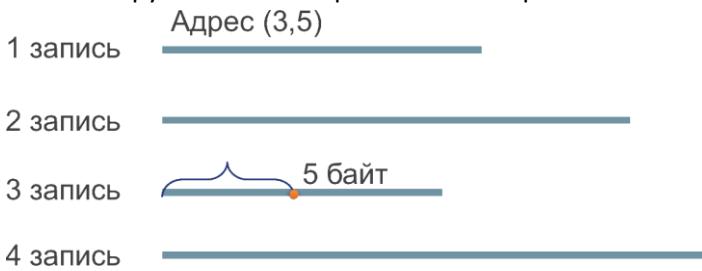
1. **Неструктурированные** последовательности байтов.
2. **Последовательность записей** фиксированной длины со своей внутренней структурой.
3. **Дерево записей** произвольной длины (каждая запись содержит поле ключа в фиксированной позиции).

Типы организации файлов

Структура файла накладывает определенные ограничения на его организацию. Под **организацией файлов** подразумевается способ расположения записей файлов во внешней памяти. Различают следующие **типы организации файлов**:

1. **Последовательная** — записи располагаются в физическом (хронологическом) порядке (1я, 2я, 3я и т.д.). Такой файл легко читается и легко записывается, но доступ к такому файлу замедленный:

для того, чтобы прочитать из 3 записи байт номер 5, то сначала придется прочитать первую запись, потом вторую и только в третий можно прочитать 5 байтов.



2. **Индексно-последовательная** — записи располагаются в логической последовательности в соответствии со значениями ключей, содержащихся в каждой записи. Доступ осуществляется прямо по ключу непосредственно к конкретной записи файла, внутри же записи доступ осуществляется последовательно. Обращение к таким файлам происходит быстрее, чем к файлам с последовательной организацией. Этот тип организации файлов используется в базах данных (БД). Для того, чтобы прочитать из 3 записи байт номер 5, то, обращение идет сразу к 3 записи не читая записи 1 и 2 (в отличии от последовательной организацией файла), а в 4 записи уже идет последовательное чтение 5-го байта.
3. **Прямая (произвольная)** — записи располагаются произвольно, доступ к ним осуществляется напрямую по их физическим адресам на запоминающих устройствах прямого доступа к памяти. Для того, чтобы прочитать из 3 записи байт номер 5, обращение будет осуществляться сразу к 5-му байту 3-й записи не читая записи 1 и 2 (в отличии от последовательной организацией файла) и не читая последовательно 4 запись (в отличии от индексно-последовательной организации файла). Это самый быстрый тип организации файлов, но, к сожалению, такие файлы не читабельны.
4. **Библиотечная** — файл, состоящий из последовательных подфайлов, называемых членами файла, начальный адрес каждого из которых хранится в директории файла. Этот тип организации файлов свойственен для таких файлов, как например, каталоги UNIX (папки или каталоги содержат имена файлов), а также характерна для оглавления библиотек, архивов и т.п. Все записи как правило имеют одинаковую длину:

| | |
|----------|---------------------------------------------------------------|
| 1 запись | Адрес первого блока файла; имя первого файла |
| 2 запись | Адрес первого блока второго файла; имя второго файла |
| 3 запись | Адрес первого блока третьего файла; имя третьего файла |
| 4 запись | Адрес первого блока четвертого файла; имя четвертого файла |

+ Средства файловой системы.

Файлами управляет ОС. Их структура, именование, защита, использование относится к той части ОС, которая называется **файловой системой**.

Файловая система содержит следующие средства:

1. **Средство доступа** — определяет конкретную организацию доступа к данным, находящимся в файлах.
2. **Средство управления файлами** — создание, копирование, перемещение, удаление и т.д.
3. **Средство управления внешней памятью** — обеспечивает распределение пространства внешней памяти для размещения файлов.
4. **Средство обеспечения целостности файлов** — гарантирует сохранность информации в файлах.

Дескриптор файла (блок управления файлом)

Информация, необходимая ОС для выполнения операций с файлами, содержится в блоке управления файлом (дескрипторе файла). Ядро ОС ведет таблицу **индексных дескрипторов файлов (ТИДФ)**, в которой содержатся все атрибуты открытых в системе файлов. Каждая запись в таблице описывает один файл и состоит из дескрипторов файлов. Эта таблица используется всеми процессами. Хранится в ОП.

Дескриптор файла — это структура данных, которая обычно хранится во внешней памяти и передается в ОП только во время открытия файла.

В **типовом дескрипторе файла** должны содержаться следующие атрибуты (необходимый минимум информации):

1. **Символическое имя файла.** Максимальная длина имени — 252 байта.
2. **Размещение во внешней памяти** (может содержаться в нескольких блоках, и все их надо указать).
3. **Тип файла** (обычные / каталоги / символические или жесткие ссылки / специальные файлы / FIFO файлы (каналы) / гнезда).
4. **Права доступа (описатель защиты),** в других системах — **средства доступа к файлу.** ОС хранит здесь информацию, с помощью каких средств она может показать данный файл.
5. **Размер файла** (в байтах).
6. **Время и дата последней модификации** (когда файл изменялся или был создан).
7. **Время и дата создания.**
8. **Время и дата последнего обращения.**
9. **Тип организации файла** (последовательная / индексно-последовательная / прямая (произвольная) / библиотечная)
10. **Диспозиция файла** (т.е. время его жизни — временный (до перезагрузки ОС — семафоры, каналы сообщений, разделяемая память и т.п.) / постоянный (постоянно хранится во внешней памяти) / рабочий (до закрытия создавшего этот файл процесса — межпроцессный канал)).

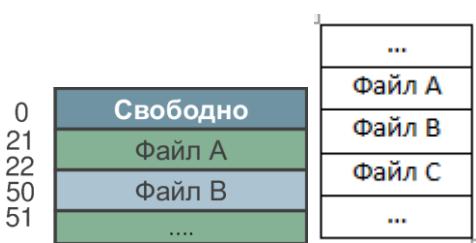
4.2 Реализация файлов. (9-1, 24-1)

Под «диском» далее подразумевается любое устройство внешней памяти. Прежде чем размещать файлы на диске, его надо разделить на блоки (отформатировать), т.к. если его делить на байты, то это повлечет за собой очень большие накладные расходы по учету занятости каждого байта. Диск делится на цилиндры, цилиндры делятся на дорожки, дорожки делятся на сектора, сектора по своей сути являются блоками. Обычно все блоки одного размера (но бывают и исключения).

Наиболее важным аспектом в реализации хранения файлов является **учет соответствия блоков диска файлам.**

Реализация файлов (схема реализации файлов)

1. **Неразрывные файлы** — набор последовательных соседних блоков диска. Файл занимает непрерывную цепочку блоков. При работе с неразрывными файлами производительность самая высокая, т.к. весь файл может быть прочитан за одну операцию. Но эта схема непригодна, если максимальный размер файла заранее неизвестен и из-за высокой фрагментации диска (неэффективное использование дискового пространства). В настоящее время эта схема реализации файлов применяется достаточно редко, но встречаются, когда надо разработать ОС специального назначения с наиболее быстрым доступом к файлам, в которых не будет большого разнообразия.



Плюс: быстрое чтение за одну операцию.

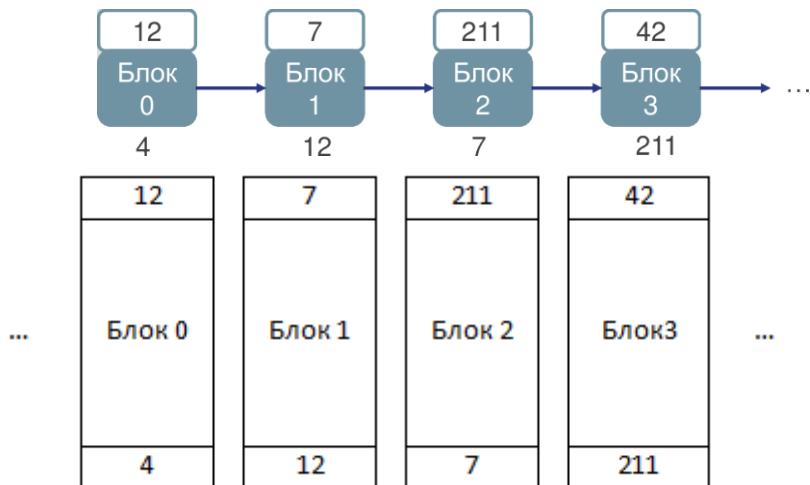
Минусы:

- Появление пустых окон при удалении файлов (проблема фрагментации). В эти окна могут не уместится новые файлы.
- Сложности с внесением изменений.
- Сложности с увеличением размера файлов.

2. **Список.** Файл хранится в виде одностороннего связного списка блоков диска. Первое слово в начале каждого блока диска — указатель на следующий блок, в последнем блоке первое слово пустое. В основной (остальной) части блока — данные файла. В каталоге ФС хранится только физический адрес первого блока файла (4 на рис.). Недостаток — медленный доступ к записям файла. Эта схема на современном этапе практически не используется, но эта схема взята за основу реализации следующей схемы.

Минусы:

- Долгий поиск.
- Если один блок испортится, сломается все.
- Долгий доступ. Чтобы получить доступ к k-му блоку файла, ОС должна прочитать первые k-1 блоков по очереди.



3. **Список с индексацией** — указатели на следующие блоки хранятся не прямо в блоках, а в отдельной таблице, загруженной в память: FAT (File Allocation Table). При такой организации прямой доступ к k-му блоку упрощен, т.к. вся цепочка ссылок (какой блок следует за каким) уже хранится в памяти и дополнительных дисковых операций не требуется. В каталоге хранится одно целое число — номер первого блока.

Минусы: таблица должна постоянно находиться в ОП. Например, небольшой диск объемом 500 МБ делится на блоки объемом 1 КБ (1024 Б). Диск будет состоять из 500 000 блоков. Так как под номер блока отводится 4 байта, FAT таблица будет объемом 2 Мб. Для большого диска таблицей FAT придется занимать почти всю оперативную память.

| | | | |
|----|-----|-----|----|
| 0 | | 0 | |
| 1 | | 1 | |
| 2 | | 2 | |
| 3 | | 3 | |
| 4 | 12 | 4 | 12 |
| 5 | | 5 | |
| 6 | | ... | |
| 7 | 211 | 12 | 7 |
| 8 | | 13 | |
| 9 | | ... | |
| 10 | | ... | |
| 11 | | | |
| 12 | 7 | | |
| 13 | | | |

Реализация такого подхода была предложена корпорацией Microsoft и часто использовалась в системах этой корпорации (MS DOS, первые версии Windows) где-то до 2000 года. Пока что можно встретить этот способ реализации в некоторых системах (FAT16, FAT32, FAT64), но время жизни этого способа подходит к завершению.

4. **Индексные дескрипторы (i-узлы, i-nodes).** С каждым файлом связана структура данных, i-узел (индексный дескриптор), содержащая все атрибуты файла, включая адреса блоков файла. ОС ведет таблицу i-узлов (индексных дескрипторов файлов), и каждая запись этой таблицы соответствует определенному файлу. Каждый конкретный i-узел находится в оперативной памяти только тогда, когда соответствующий файл открыт. Если размер i-узла n байт (в UNIX — 64 байта), а открыто k файлов, то таблица займет в оперативной памяти $n \cdot k$ байт. Эта величина намного меньше размера FAT таблицы и не зависит от объема диска. Этот способ реализации файлов взят за основу в UNIX-подобных ОС. Реализация i-узлов позволяет вмонтировать в контур файловых систем UNIX любую реализацию файловой системы, в т.ч. NTFS и FAT.

5. **New Technology File System (NTFS).** Файл состоит из атрибутов, каждый из которых представляется набором байтов. Большинство файлов имеет три атрибута:

- имя файла;
- идентификатор файла (64 бита);
- поток данных файла, т.е. блоки, в которых файл располагается (максимальная длина потока — 2^{64} байта ~ 20 ЭБ);

Если файл большой, то у него может быть несколько потоков данных, тогда добавляется новый атрибут — второй поток и т.д.

Диск делится на тома (дисковые разделы — C:, D:, E;...). Каждый том организован как линейная последовательность блоков (кластеров). Обращение к блокам осуществляется по их смещению от начала тома. Основной структурой данных в каждом томе является таблица MFT (Master File Table), состоящая из записей размером в 1 КБ. Каждая запись MFT описывает один файл или каталог.

Максимальное количество записей в таблице MFT — 2^{48} . Каждая запись таблицы содержит атрибуты файла:

- Имя файла (256 байт).
- Идентификатор файла (64 байта).
- Номера блоков диска, содержащих данный файл.

(Если файл слишком большой, то создается дополнительная запись, чтобы вместить все блоки файла).

С помощью таблицы MFT ОС следит за состоянием файлов и регламентирует обращение пользователей к файлам. Технология NTFS была впервые представлена в 1993 г. корпорацией Microsoft, начиная с Windows NT 3.1 является ФС по умолчанию для ОС семейства Windows NT.

+ Реализация каталогов

Прежде чем работать с файлом, его нужно открыть. При открытии файла, ОС оперирует полным именем файла, чтобы найти запись о файле в каталоге.

Запись в каталоге содержит информацию, необходимую для нахождения блоков диска. Это может быть:

- дисковый адрес всего файла для неразрывных файлов;
- номер первого блока для обеих схем списков;
- номер *i*-го узла.

В первых трех случаях запись в каталоге должна содержать информацию об атрибутах файла. *i*-й узел такую информацию уже содержит и это дополнительное преимущество такой схемы реализации.

4.3-4.4 Файлы в файловой системе UNIX-подобных ОС

Файловая система UNIX имеет многоуровневую иерархическую структуру и предназначается для разделения внешней памяти на файлы. С ней связаны и вопросы ввода-вывода, и информационные связи между процессами, и многое другое.

Отличительной особенностью ОС UNIX является то, что понятие файла в ней максимально унифицировано. Под файлами понимаются любые источники и потребители информации. Это и традиционные файлы, и внешние устройства и процессы.

С понятием файла связаны все ресурсы в UNIX-системах. Тип файла можно узнать с помощью утилиты

ls -l <имя_файла> (первый символ в выводе)

```
ubuntu@ubuntu:~$ ls -l a.txt
-rwxrwxrwx 1 ubuntu ubuntu 8 Mar 24 18:40 a.txt
ubuntu@ubuntu:~$
```

Флаг **-l** подразумевает вывод подробной информации.

или утилиты

file <имя_файла>.

```
ubuntu@ubuntu:~$ file test
test: directory
ubuntu@ubuntu:~$
```

Утилита **find** не только определяет стандартные типы файлов, но и распознает их форматы.

Типы файлов в файловой системе ОС UNIX.

1. Обычные (регулярные), первый символ в выводе для **ls** — символ тире (-).
2. Каталоги (d).
3. FIFO (каналы) (p).
4. Специальные (b — блок-ориентированные (устройства типа дисковых накопителей, информация в которых передается блоками),
c — байт-ориентированные (устройства типа клавиатуры, информация с которых передается в виде байтов)).
5. Гнезда (s) (т.е сокеты), это файлы, которые используются для обмена информации по сети.
6. Символьная ссылка на файл (l).

Примечание: жесткую ссылку отдельным типом файла называть нельзя, т.к. жесткая ссылка — это обычный (регулярный) файл.

Регулярные (обычные) файлы

Содержат информацию, определенную пользователем или формирующуюся в результате работы прикладных или системных программ. Никаких ограничений на информацию, хранящуюся в обычных файлах, не накладывается.

Различают **текстовые и двоичные (бинарные) обычные файлы**. Бинарные файлы — это либо файлы, приготовленные для выполнения, либо не исполняемые — картинки, рисунки, видео и т.п. Т.е. обычный файл может иметь статус выполнимого — такой файл будет рассматриваться системой как:

- Программа написанная на командном языке.
- Программа на машинном языке, готовая к выполнению.

Поскольку в UNIX-подобных ОС файлу предоставляются права на выполнение через настройки уровня доступа в файловой системе, а не через его расширение, то расширения файлов в UNIX-подобных ОС не играют никакой роли. Исключением являются тексты программ на некоторых языках программирования. Так, если вы пишете программу на языках С или С++, то нужно добавить к файлу расширение ".c" или ".cpp" для того, чтобы транслятор обработал этот файлы. Обычно пользователи добавляют расширения для того, чтобы сразу было понятно, какой именно это файл.

Обычные файлы могут иметь **жесткие и мягкие (символьные) ссылки**, например:

1. **`ln a.c b.c`** (жесткая ссылка на файл a.c). **Жесткая ссылка** — это альтернативное имя файла (в текущем или в любом каталоге) в том же разделе диска. Т.е. файл занимает одно место хранения, но для него существуют два имени. К файлу можно получить доступ через множество разных имен, известных как жесткие ссылки. Если под одним именем удалить этот файл, то он все равно не удалится, потому что у него останется еще одно имя. Количество жестких ссылок файла сохраняется на уровне файловой системы в метаинформации индексного дескриптора. Файлы с нулевым количеством ссылок перестают существовать для системы и, со временем, будут перезаписаны физически. В файловых системах UNIX-подобных ОС при создании файла на него автоматически создается одна жесткая ссылка (на то место файловой системы, в котором файл создается). Дополнительную ссылку в UNIX можно создать с помощью команды **`ln`** (см. выше). Все ссылки одного файла равноправны и неотличимы друг от друга — нельзя сказать, что файл существует в таком-то каталоге, а в других местах есть лишь их копии. Удаление ссылки приводит к удалению файла лишь в том случае, когда это была последняя ссылка, любая из созданных, то есть все остальные жесткие ссылки на него уже удалены.

Теперь немного о жестких ссылках на файл в другом каталоге. Файл может обозначаться несколькими путевыми именами, если пользователь создаст одну или несколько ссылок на него с помощью команды **`ln`**:

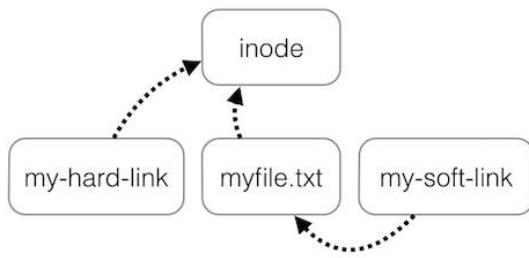
`ln /usr/bin/who /tmp/ww`

2. **`ln -s a.c b.c`** (мягкая/символьная/символическая ссылка на файл a.c.). **Символическая ссылка** на файл отличается от жесткой тем, что помимо альтернативного имени файла создается новый индексный дескриптор файла (новую запись в таблице индексных дескрипторов файлов (ТИДФ)). Т.е. файл характеризуется не одной записью в таблице индексных дескрипторов, а двумя записями. Символическая ссылка не изменяет счетчик значений жестких ссылок соответствующего индексного дескриптора.

С помощью мягкой ссылки можно ссылаться:

- на файл в другой файловой системе или даже в сети;
- на каталог (но чаще всего пользователизываются на регулярные (обычные) файлы);
- на несуществующий файл (в последнем случае при попытке открыть его должно выдаваться сообщение об отсутствии файла). Даже если при создании символической ссылки (используя ключ `-s`) обозначаемый «файл» окажется несуществующим, символическая ссылка всё равно будет создана (с именем `b.c`). Ссылка, указывающая на несуществующий файл, называется висячей или битой.

Доступ к файлу можно получить с помощью различных ссылок, указывающих на этот файл, называемый мягкой ссылкой. См. также [«В чем разница между жесткой ссылкой и файлом?»](#)



Для создания обычных файлов используются утилиты текстовых редакторов (**vi**, **vim**, **nano**, **edit**, ...), утилита **touch**, системные и прикладные программы (компиляторы, фильтры и т.д.).

Для просмотра (чтения) обычных файлов используются утилиты **cat**, **more**, **pr**, **head** (вывод первых строк файла), **tail** (вывод последних строк файла) и др.

Для удаления обычных файлов используются утилиты **rm** или **unlink**.

Каталоги

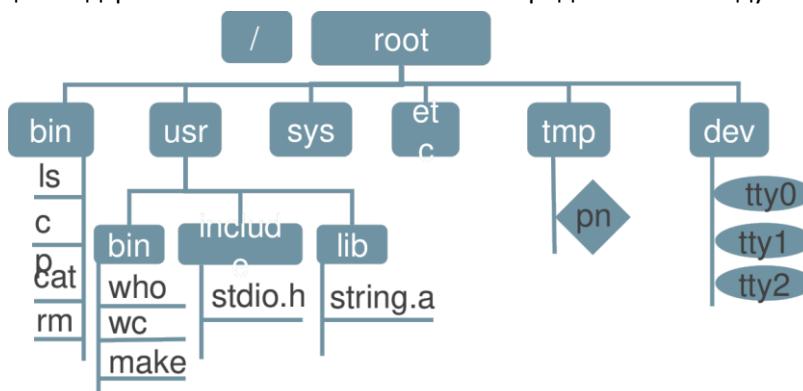
Файл, который содержит информацию о других файлах в ФС в виде несортированного набора записей длиной в 256 Б (4 Б в записи отводятся под номер i-узла (индексного дескриптора), 252 Б под имя). Таким образом, UNIX-подобные ОС допускают размер имени файла в 252 символа.

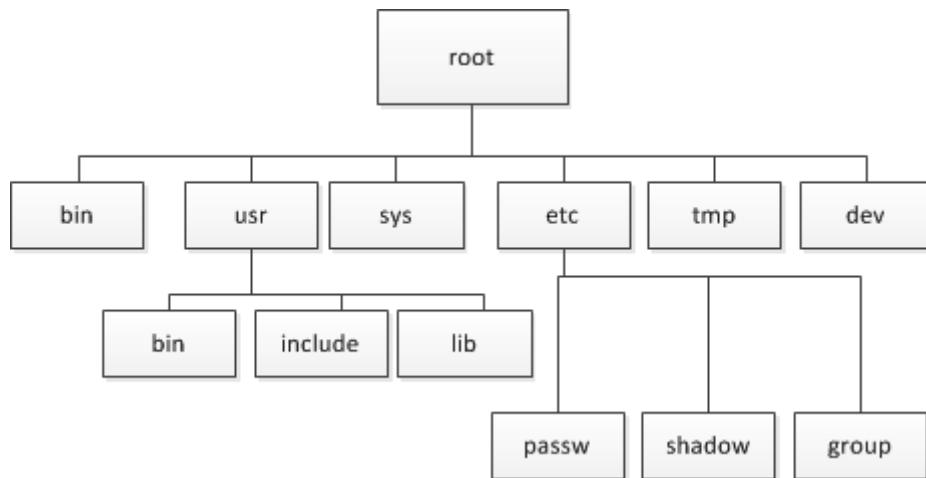
Каталоги предоставляют пользователям средства для организации их файлов в некоторую иерархическую структуру, основанную на взаимосвязи файлов и направлении их использования.

- Каталог, который не является подкаталогом ни одного другого каталога, называется **корневым** (/root или просто /). Это значит, что этот каталог находится на самом верхнем уровне иерархии всех каталогов.
- **Подкаталог** — каталог файлов, созданный внутри другого каталога.
- Имя файла, указанное начиная с корневого каталога, называется **полным именем файла** (или жесткая ссылка на файл). Например, полное имя файла для утилиты **wc**: /usr/bin/wc
- Каталог, в котором пользователь находится в данный момент времени, называется **текущим** каталогом.
- Если используется неполное имя файла, файл всегда будет искааться в текущем каталоге.
- Каталог ОС UNIX считается **пустым**, если он не содержит никаких других файлов кроме ссылок на текущий (.) и родительский (..) каталоги. Такой каталог можно удалить утилитой **rmdir**.

Существует отличие файловой системы UNIX от MS-DOS, WINDOWS и т.п. заключается в возможностях монтирования другой файловой системы в топологию файловой системы UNIX. В файловой системе имеется выделенный **корневой** каталог. Каждый следующий уровень состоит из каталогов и других файлов, подчиненных корневому каталогу. Поэтому легко адресоваться к любому файлу, указав путь, начиная с корневого каталога.

Общий вид файловой системы UNIX можно представить в следующем виде:





Наиболее популярные и часто посещаемые каталоги, в которых находятся команды, с помощью которых пользователь может манипулировать ОС, может просматривать различную информацию в ОС:

- /bin — содержит команды операционной системы ядра, в нем расположена основная масса самых важных утилит, которую пользователь может использовать для работы с системой.
- /usr/bin — в нем находится оставшаяся часть стандартных утилит.

Другие важные каталоги:

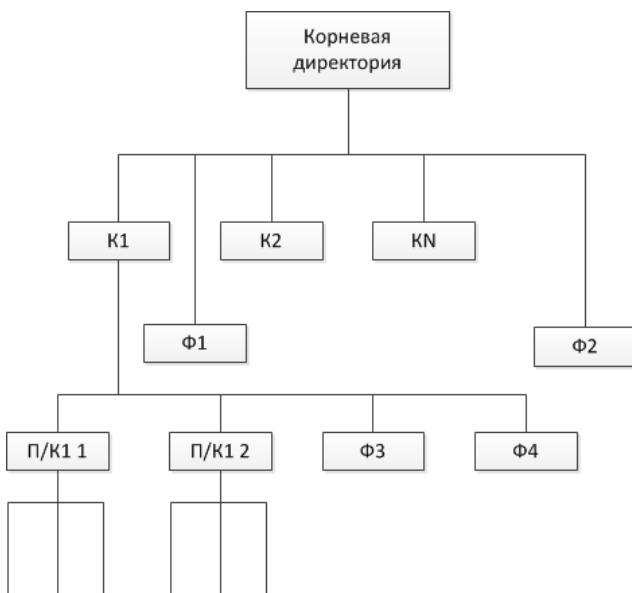
- /root — домашний каталог суперпользователя (часто просто /).
- /usr — в нем хранится большинство стандартных программ (/usr/bin) и другие полезные компоненты, в частности интерактивная документация, библиотеки (/usr/lib) и файлы заголовков, предназначенные для компиляции C -программ (/usr/include).
- /sys — содержит интерфейсы разных ядер (Linux).
- /etc — содержит критические системные файлы и файлы конфигурации системы.
- /tmp — для временных файлов, которые могут удаляться при перезагрузке.
- /dev — в нем находятся файлы устройств: дисков, принтеров, псевдотерминалов и т.д.

Утилиты для работы с каталогами:

- Для создания каталогов используется утилита **mkdir**.
- Для удаления каталогов используются утилиты **rmdir** (удаление пустого каталога), **rm** (удаление каталога с файлами).

+ Топология файловой системы (26-1, 24-1)

Топология файловой системы — древовидная структура, во главе которой находится корневой каталог.



FIFO файлы (каналы)

Предназначены для создания/открытия временного буфера (канала), обеспечивающего взаимодействие двух или нескольких процессов посредством записи данных в этот буфер и чтения данных из буфера. Это односторонние (полудуплексные) файлы. FIFO файлы работают по схеме «first-in-first-out» (первый пришел — первый ушел). Буфер, связанный с FIFO файлом, создается, когда один из процессов открывает этот файл для чтения или записи. Буфер удаляется, когда все процессы, связанные с FIFO файлом, закрывают все свои ссылки на него (межпроцессный канал) или когда сам пользователь удаляет его (именованный канал и канал сообщений).

В какой области памяти создаются файлы разных видов каналов:

- в ОП в пространстве пользователя — межпроцессный канал;
- во внешней памяти — именованный канал;
- в ОП в пространстве памяти ядра ОС — канал сообщений.

Все эти файлы представляют собой каналы, единственная особенность — межпроцессный канал и именованный каналы — байтовые (3 входящих «сообщения» на выходе сливаются в одно большое сообщение), а канал сообщений — не байтовый канал (3 входящих «сообщения» на выходе вернут также 3 сообщения, причем сообщения из канала сообщений можно извлекать выборочно).

Каждый вид каналов создается с помощью своих средств.

Межпроцессный канал:

- `pipe()` — СВ для создания и открытия межпроцессного канала.
- Межпроцессный канал — файл с рабочей диспозицией, он удаляется, когда все процессы, связанные с FIFO файлом, закрывают все свои ссылки (дескрипторы) на него или когда завершится процесс, создавший этот файл.

Именованный канал:

- `mkfifo, mkfifo()` — утилита, СВ для создания/открытия именованного (временного) канала:
`mkfifo fifo.file`
- FIFO файл также может быть создан в UNIX с помощью `mknod` с атрибутом `r`:
`mknod fifo.file r`

Эти две команды выполняют идентичные операции, за одним исключением. Команда `mkfifo` предоставляет возможность для изменения прав доступа к файлу FIFO непосредственно после создания. При использовании `mknod` будет необходим вызов команды `chmod`.

- Для удаления именованного канала используется утилита `rm` (он может быть удален подобно обычному файлу). Именованный канал существует в системе и после завершения процесса. Он должен быть «отсоединен» или удален, когда уже не используется. Процессы обычно подсоединяются к каналу для осуществления взаимодействия между ними. Именованный канал — файл с временной диспозицией, он может работать, пока не будет удален или пока ОС не будет перезагружена.

Канал сообщений:

- `msgget()` — СВ для создания/открытия канала сообщений.
- Для просмотра канала сообщений используется утилита `ipcs -q`;
- Для удаления канала сообщений используется утилита `ipcrm -q`;
- Канал сообщений — файл с временной диспозицией, он будет жить до тех пор, пока не будет удален или пока ОС не будет перезагружена.

Специальные файлы

Предназначены для организации взаимодействия с УВВ (устройствами ввода-вывода). С каждым внешним устройством связан свой специальный файл. Бывают **байт-ориентированными и блок-ориентированными**. С ними можно работать как с обычными файлами, т.е. прикладная программа может выполнять операции чтения и записи со специальным файлом точно так же, как и с обычным файлом. ОС будет автоматически

вызывать соответствующий драйвер устройства для выполнения фактической передачи данных между физическим устройством и данной программой. Т.е. с точки зрения пользователя работать с УВВ становится очень просто: открыли файл и записали/прочитали информацию.

Создавать специальные файлы может только администратор UNIX-подобной ОС. Создаются с помощью утилиты **mknod**.

Пример создания специального файла:

mknod /dev/crk c 115 5 – байт-ориентированное устройство.

mknod /dev/brk b 121 15 – блок-ориентированное устройство.

где:

- Имя файла **/dev/crk** и **/dev/brk**.
- Ориентацию специального файла определяет **c** (байт) и **b** (блок).
- Старший номер (тип устройства) — 115 и 121 — номер записи в таблице драйверов, которую ведет ОС (таблица ядра содержит список всех драйверов, известных системе).
- Младший номер (номер устройства) — 5 и 15 — передается в драйвер устройства при его вызове как 1-й внешний аргумент. Этот номер сообщает драйверу, с каким конкретно физическим устройством он взаимодействует. Например, если в системе два абсолютно одинаковых диска, то старший номер (тип устройства) у них будет совпадать, а различаться будут младшие номера (номера устройств). Иначе говоря, этот номер используется для того, чтобы отличить одно **одинаковое устройство (одного типа)** от другого.

Гнезда (sockets)

Файлы, предназначенные для обмена информацией через сеть между процессами, работающими на разных компьютерах.

Для каждого файла (гнезда) назначается **тип**:

- **виртуальный канал (virtual circuit)** — данные передаются последовательно с достаточной степенью надежности (с проверкой, обычно с использованием протокола TCP);
- **дейтаграмма (datagram)** — последовательность пересылки данных не выполняется, надежность передачи данных — низкая (без проверки, обычно с использованием протокола UDP, актуально при передаче потоковой аудио и видео информации, когда потеря нескольких битов никак не повлияет визуально или аудиально).

Различают гнезда:

- **с установлением соединения** (адреса гнезд отправителя и получателя выясняются заранее до передачи данных между ними);
- **без установления соединения** (адреса гнезд отправителя и получателя передаются с каждым сообщением, посыпаемом от одного процесса другому).

Системные вызовы для работы с гнездами:

- **socket()** — создает гнездо заданного типа, с указанным протоколом, в указанном семействе адресов.
- **bind()** — присваивает гнезду имя, т.е. осуществляет привязку к адресу.
- **listen()** — вызывается в серверном процессе для прослушивания клиентского гнезда, ориентированного на установление соединения типа виртуальный канал. Т.е. этот СВ предназначен для того, чтобы сервер ждал, когда к нему обратится клиентский процесс, работающий удаленно.
- **connect()** — вызывается в клиентском процессе для установления соединения с серверным гнездом.
- **accept()** — вызывается в серверном процессе для установления соединения с клиентским гнездом, которое осуществило СВ **connect()**. СВ нужен для того, чтобы понять, с каким клиентом организовать связь.

- **recv()** — читает сообщение из гнезда.
- **recvfrom()** — читает сообщение из гнезда, адрес которого указывается в аргументах **recvfrom()**.
- **send()** — передает данные в заданное гнездо.
- **sendto()** — записывает данные в гнездо, адрес которого указывается в аргументах **sendto()**.
- **shutdown()** — закрывает соединение между «серверным гнездом» и «клиентским гнездом».
- **close()** — закрывает файл гнезда.

Утилита которую можно использовать для просмотра гнезд — **netstat**.

+ Последние изменения в файловой системе ОС UNIX

1. *Поиск файлов в каталогах производится линейно. Для увеличения производительности системы было добавлено кэширование имен. Прежде чем искать имя в каталоге, система проверяет его наличие в кэше, если есть в кэше — в каталоге можно не искать.*
2. *Разбиение диска на группы цилиндров (на группы блоков). У каждой группы создается супер-блок, индексные дескрипторы и блоки данных. Суть: хранить индексные дескрипторы и данные ближе друг к другу, чтобы снизить время, затрачиваемое жестким диском на перемещение головок.*
3. *Использование не одного, а двух размеров блоков. Для файлов большого размера — блоки 1024 байт; для файлов маленького размера — 512 байт. Наличие блоков двух размеров позволяет обеспечивать эффективность операций чтения\записи для больших файлов и эффективное использование дискового пространства для небольших файлов.*

5. Поддержка и реализация файлов в ОС UNIX. Системные вызовы и утилиты для получения информации о файле

5.1 Индексный дескриптор файла в ОС UNIX. Таблица файлов. (6-2)

С каждым файлом в ОС UNIX связан **индексный дескриптор**. Это структура данных (64 байта), которая имеет следующие атрибуты:

- **права доступа к файлу** для владельца, группы и прочих;
- **количество (счетчик) жестких ссылок** на файл;
- **UID идентификатор владельца**;
- **GID идентификатор группы**;
- **размер файла в байтах**;
- **время последнего доступа** к файлу (когда программа или пользователь читала этот файл);
- **время последней модификации** файла (когда файл изменялся или был создан);
- **время последнего изменения прав доступа**;
- **системный номер индексного дескриптора файла** (по этому номеру сам индексный дескриптор в таблице во внешней памяти и вызывается — см. последний абзац);
- **идентификатор файловой системы**, в которой находится файл;
- **тип файла** (обычные / каталоги / символические или жесткие ссылки / специальные файлы / FIFO файлы (каналы) / гнезда);
- **тип организации файла** (последовательная / индексно-последовательная / прямая (произвольная) / библиотечная);
- **физический адрес на диске** (обычно указатель на адрес первого блока, в системах UNIX дальше указания на блоки файлов обычно хранятся в связанным списке);
- **диспозиция файла (т.е. время его жизни** — временный (до перезагрузки ОС — семафоры, каналы сообщений, разделяемая память и т.п.) / постоянный (постоянно хранится во внешней памяти) / рабочий (до закрытия создавшего этот файл процесса — межпроцессный канал)).

Для каждой файловой системы (ОС UNIX поддерживает неограниченное их количество) формируется свой идентификатор, и создается таблица индексных дескрипторов, в которой хранится информация обо всех файлах.

Каждая запись в таблице индексных дескрипторов содержит все перечисленные атрибуты файла и определяется по идентификатору файловой системы и номеру индексного дескриптора. Идентификатор присваивается файловой системе при выполнении команды **mount**.

Всякий раз, когда создаются новый файл, ядро ОС UNIX создает новую запись в таблице индексных дескрипторов для сохранения информации о нем. Кроме того, ядро ОС добавляет имя файла и номер дескриптора в соответствующий каталог.

Символическое имя файла отсутствует в индексном дескрипторе файла! Это сделано для того, чтобы на файл можно было сделать неограниченное количество жестких или символьных ссылок. Символическое имя файла хранится только в каталогах, т.е. каждая запись каталога содержит имена файлов.

Таблицы индексных дескрипторов содержатся в соответствующих файловых системах на диске (т.е. во внешней памяти), но ядро ОС UNIX ведет их копии в ОП. В этих копиях в ОП содержится информация только о тех файлах, которые открыты в данный момент, т.к. вся таблица целиком может и не поместиться в ОП. Если файл не открыт, то информация о нем в таблице-копии, которая ведется в ОП, отсутствует. После того, как открытый файл закрывается, информацию о нем, которая хранится в ОП необходимо перенести на диск (во внешнюю память), именно для того, чтобы знать, в какое место ее перенести, указывается системный номер индексного дескриптора файла.

Итак, ядро ОС поддерживает таблицу индексных дескрипторов файлов (ТИДФ), в которой содержатся все атрибуты открытых в системе файлов. Используется всеми процессами.

| N | Права доступа к файлу | Размер файла (в байтах) | Количество (счетчик) жестких ссылок | UID – идентификатор владельца файла | GID – идентификатор группы файла | Идентификатор файловой системы в которой находится файл | Время последнего доступа к файлу | Время последней модификации | Время последнего изменения прав доступа | Диспозиция файла (временный / постоянный / рабочий) | Тип организации файла | Физический адрес (адрес первого блока) | Системный номер индексного дескриптора файла | Тип файла |
|-----|-----------------------|-------------------------|-------------------------------------|-------------------------------------|----------------------------------|---------------------------------------------------------|----------------------------------|-----------------------------|-----------------------------------------|-----------------------------------------------------|-----------------------|----------------------------------------|----------------------------------------------|-----------|
| 0 | | | | | | | | | | | | | | |
| 1 | | | | | | | | | | | | | | |
| 2 | | | | | | | | | | | | | | |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

(Здесь же рассказать весь следующий вопрос!)

5.2 Поддержка файлов ядром ОС UNIX . (4-2)

Выше уже говорилось, что ядро ОС поддерживает таблицу индексных дескрипторов файлов (ТИДФ), в которой содержатся все атрибуты открытых в системе файлов. Используется всеми процессами. Еще раз напомним, что эта таблица — копия части таблицы индексных дескрипторов, содержащейся в соответствующей файловой системе на диске (во внешней памяти).

Ядро ОС UNIX ведет также таблицы файлов, в которых отслеживаются все открытые в системе файлы.

И каждый вновь создаваемый процесс ведет собственную таблицу пользовательских дескрипторов открытых файлов, где регистрируются все файлы, открытые этим процессом. Таблица содержится в контексте процесса и может подвергаться свопингу, т.е. выгрузке во внешнюю память, когда процесс выгружается из ОП. Эта т.н. ТПДОФ процесса (таблица пользовательских дескрипторов открытых файлов процесса) содержит всего два поля:

- имя файла;
- ссылка на таблицу файлов (ТФ) системы, которую ведет ядро ОС (о ней чуть ниже).

| | Имя файла | № ТФ |
|---|-----------|------|
| 0 | /dev/tty | 10 |
| 1 | /dev/tty | 21 |
| 2 | /dev/tty | 6 |
| 3 | a.txt | 4 |
| 4 | ... | ... |

Ядро ОС поддерживает также таблицу файлов (ТФ), в которой отслеживаются все открытые в системе файлы. Используется всеми процессами. Такое разделение на две таблицы связано с тем, что в ТИДФ находятся в основном информация о файлах и их атрибуты. В ТФ же хранятся следующие характеристики открытых файлов:

- Режим открытия файла (чтение, запись, чтение и запись и т.п.);
- Указатель чтения/записи открытого файла;
- Счетчик ссылок — сколько дескрипторов файлов из процессов ссылаются на данную запись в таблице открытых файлов (ТФ) в системе;
- Ссылка на номер записи в таблице индексных дескрипторов открытых файлов (ТИДФ), которая содержит все атрибуты индексного дескриптора файла, которые характеризуют данный файл.

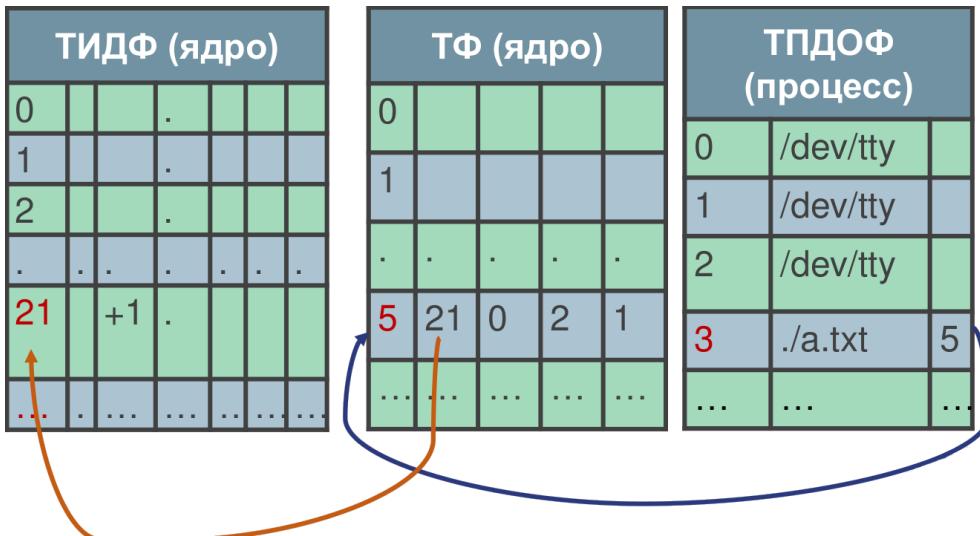
| | № ИД | Указатель на текущую позицию в файле | Режим открытия файла | Счетчик ссылок |
|-----|------|--------------------------------------|----------------------|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | 241 | 0 | 0 | 1 |
| ... | ... | ... | ... | ... |

Итак, с помощью этих трех таблиц ОС поддерживает взаимодействие с файлами в UNIX-подобных системах:

- **Таблица индексных дескрипторов файлов (ТИДФ)**, в которой содержатся все атрибуты открытых в системе файлов. Используется всеми процессами. Хранится в ОП. Является частичной копией таблицы индексных дескрипторов, которая содержится в соответствующих файловых системах на диске (т.е. во внешней памяти).
- **Таблица файлов (ТФ) файловой системы**, в которой отслеживаются все открытые в системе файлы. Используется всеми процессами. Хранится в ОП.
- **ТПДОФ процесса (таблица пользовательских дескрипторов открытых файлов процесса)**. Содержится в контексте процесса, может подвергаться свопингу, когда процесс выгружается из ОП.

Пример:

```
int fd = open("a.txt", 2);
```



Когда процесс вызывает системный вызов `open/creat`, чтобы открыть файл для чтения или записи, ядро преобразует имя файла в индексный дескриптор файла.

Если индексный дескриптор файла доступен процессу (есть право доступа), ядро ищет в таблице пользовательских дескрипторов файлов первую свободную строку. Номер этой строки будет возвращен процессу, как пользовательский дескриптор открытого файла (номер 3 на рис.).

Далее ядро просматривает таблицу файлов и ищет там первую свободную строку. В записи таблицы пользовательских дескрипторов файлов процесса делается ссылка на найденную позицию в таблице файлов (номер 5 на рис.).

В записи таблицы файлов производится ссылка на ту запись в таблице индексных дескрипторов файловой системы, в которой хранится индексный дескриптор (номер 21 на рис.) этого файла. В записи таблицы файлов формируется указатель текущей позиции (чтения/записи) в файле (ячейка, содержащая 0 на рис.). В запись таблицы файлов заносится информация, о том, в каком режиме открыт файл (ячейка, содержащая 2 на рис.).

говорит о том, что файл открыт на чтение и на запись) и значение счетчика ссылок в записи таблицы файлов устанавливается в соответствии с тем, сколько дескрипторов файлов из процесса обращается к данной записи (ячейка, содержащая 1 на рис.).

Значение счетчика ссылок индексного дескриптора файла увеличивается на единицу (ячейка, содержащая +1 на рис.).

Ядро будет использовать пользовательский дескриптор открытого файла, как индекс в таблице пользовательских дескрипторов файла процесса для поиска элемента или строки в таблице файлов, соответствующих открытому файлу.

Затем ядро проверяет данные записи в таблице файлов, чтобы убедиться в том, что файл открыт в соответствии с режимом доступа. Ядро использует указатель из записи в таблице файлов для доступа к записи индексного дескриптора файлов. Оно использует указатель чтения/записи файла из таблицы файлов, чтобы определить с какого элемента должны происходить чтение или запись.

Ядро проверяет вид файла в записи индексного дескриптора и вызывает соответствующие драйверы для того, чтобы фактически начать обмен данными с физическим устройством.

5.3-5.4 Физическая и логическая организация файловой системы ОС UNIX. (16-2)

В ОС UNIX физическая и логическая структура файла не совпадают. Логически файл представляется непрерывной цепочкой блоков, а физически — списком блоков, которые могут быть разбросаны по всему пространству внешней памяти (дисковому пространству).

Недостатки такой организации проявляются в дополнительных системных расходах на ведение списка блоков и поиск информации. Но такие положительные моменты как возможность неограниченного расширения файлов, унификация вопросов связанных с вводом-выводом оказались более весомыми для определения способа организации файловой системы.

Суть логической организации — файл есть непрерывная последовательность байтов и к нему можно непрерывно прямо обращаться с помощью, например, СВ **Iseek()**. При открытии файла указатель текущей позиции всегда устанавливается на первый байт файла. Но если файл изменять не требуется, а надо дописать в него, то указатель текущей позиции с помощью СВ **Iseek()** можно переместить на конец файла и дописывать в него информацию или на любой другой байт, который потребуется для выполнения заданных вами действий.

Физическая организация файловой системы представляет собой совокупность блоков, расположенных во внешней памяти (на диске).

| 0 | 1 | 2 isize/(8,16,32...) +2 | | |
|-------------------------|------------|--------------------------------|---------------|----------------------------------|
| Блок начальной загрузки | Супер-блок | Индексные дескрипторы файлов | Данные файлов | Свободные блоки, косвенные блоки |
| 0 fsize | | | | |

1. **Нулевой блок — Блок начальной загрузки ОС.** В нем расположен адрес, с которого осуществляется загрузка ОС, или адрес программы-загрузчика, которая предоставляет возможность загрузить любую ОС по вашему выбору.
2. **Первый блок — Супер-блок,** который **содержит заголовок файловой системы.** В заголовке находится:
 - информация о размере файловой системы (**fsize**) в блоках;

- числе/количество индексных дескрипторов (isize) в файловой системе, т.е. максимальное количество файлов в разделе ФС;
- ссылка на список свободных блоков.

3. **Начиная со второго, несколько блоков, содержащих индексные дескрипторы файлов.** Количество таких блоков определяется количеством индексных дескрипторов (количество блоков = $isize/(blocksize/idsize)$, где idsize — размер одного индексного дескриптора, который обычно равен 64 байтам). В зависимости от размера блока, блок может вмещать 8, 16, 32 и т.д. индексных дескрипторов. В ОС UNIX имя файла отсутствует в индексных дескрипторах. Индексный дескриптор корневого каталога имеет номер 2.

4. **Все оставшиеся блоки используются под данные или образуют список свободных блоков и косвенных блоков. Косвенные блоки — свободные блоки, которые вы зарезервировали (забронировали), но не используете в данный момент (по сути свободные и их всегда можно переставить в список свободных).**

Для того, чтобы получить быстрый доступ к данным файла, желательно индексные дескрипторы файлов держать поближе к блокам данных, что позволяет блоку головок магнитного диска значительно быстрее считывать информацию из файлов. Для этого в современных ФС UNIX-подобных ОС диски большого размера обычно делят на разделы, называемые группами цилиндров или группами блоков. Каждая группа блоков имеет свой супер-блок, свои блоки с индексными дескрипторами, блоки с данными и свободные блоки.



Для обеспечения быстрого доступа к файлам и эффективности работы всей файловой системы в ОП (основной памяти или области памяти ядра ОС) должны располагаться следующие компоненты ОС UNIX:

- Супер-блок. В супер-блоке — состояние файловой системы, в том числе и информация о свободном месте. При запуске ОС супер-блок считывается в память и все изменения файловой системы вначале находят отображение в копии супер-блока, находящейся в ОП, и записываются на диск только периодически. Это позволяет повысить производительность системы, так как многие пользователи и процессы постоянно обновляют файлы.
- Копия таблицы индексных дескрипторов файлов (ТИДФ) — каждая запись в ней определяет информацию, необходимую для доступа к открытому файлу.
- Таблица файлов (ТФ) файловой системы, каждый элемент которой содержит ссылку на некоторый адрес в таблице индексных дескрипторов файлов (ТИДФ) и указатель текущей позиции, указывающий на очередной байт файла, подлежащий обработке.

5.5 Системные вызовы и утилиты, позволяющие получить информацию о файле

Системные вызовы, позволяющие управлять файлами:

stat() — позволяет получать атрибуты индексного дескриптора открытого файла. Ниже представлен пример получения размера файла main.cpp в байтах. Сначала файл открывается для чтения, далее индексный дескриптор файла читается в переменную fp, затем с помощью СВ **stat()** в переменную buff запишется

структурированная запись из индексного дескриптора файла. Из переменной `buff` затем получаются все атрибуты указанного файла, например размер файла в байтах, который затем выводится на экран монитора.

```

1 #include <stdio.h>
2 #include <sys/stat.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <stdlib.h>
6 #include <fcntl.h>
7 int main(void)
8 {int fp; struct stat buff;
9 if((fp = open("main.cpp", 0)) == -1) {
10 | printf("Cannot open file.\n");
11 | exit (1);
12 }
13 stat("main.cpp", &buff); // Заполнение структуры типа stat
14 printf("Size of the file is: %ld\n", buff.st_size);
15 lseek(fp, 10, 0);
16 close(fp);
17 return 0;
18 }
19

```

lseek() — перемещает указатель чтения/записи открытого файла. При открытии файла указатель текущей позиции всегда устанавливается на первый байт файла. Но если файл изменять не требуется, а надо дописать в него, то указатель текущей позиции с помощью СВ **lseek()** можно переместить на конец файла и дописывать в него информацию или на любой другой байт, которые потребуются для выполнения заданных вами действий.

access() — проверяет права доступа к файлу (можно ли читать в этот файл, писать в этот файл, обращаться в этот файл). Полезно перед обращением к файлу проверить, а можете ли вы работать с этим файлом? Если вы не можете работать с файлом, то либо нужно поменять права доступа, либо совершить другие действия, чтобы можно было работать с данным файлом.

mknod() — создает новый файл, каталог или специальный файл. Обычно используется для создания специальных файлов. Для создания каталогов используется более современный СВ **mkdir()** — см. ниже.

lchown() — изменяет владельца файла у символьской ссылки на файл (принимает символьскую ссылку).

fchown() — изменяет владельца файла (принимает пользовательский индексный дескриптор открытого файла).

open() — открывает файл в заданном режиме открытия (чтение, запись, чтение и запись и т.п.) или создает новый файл.

creat() — создает файл с указанными правами доступа и открывает его на запись. Причем вне зависимости от того, какие права доступа вы даете файлу, если создаваемый файл не существует, то СВ **creat()** всегда создаст файл с любыми правами доступа и откроет его на запись, даже если при создании этого файла вы не даете прав владельцу на запись в этот файл. По сути это оболочка к СВ **open()**. Если файл с таким именем уже существует, то, в зависимости от прав доступа, вы можете и не пересоздать этот файл. Т.е. если права доступа запрещают осуществлять запись в уже существующий файл, то **creat()** не создаст этот файл заново и не откроет его на запись, а вернет **-1**.

popen() — комбинированный СВ, вызывающий несколько СВ: СВ **pipe()** создает межпроцессный канал, затем СВ **fork()** порождает новый дочерний процесс; в контексте нового процесса выполняет команду, заданную в аргументах **popen()** и записывает результат команды в межпроцессный канал. Т.е. с помощью СВ **popen()** можно получить результат выполнения утилиты или какой-либо программы, которую вы хотите запустить и получить его результаты в рамках одного процесса.

pclose() — закрывает канал, открытый **popen()**, и ждет завершения порожденного **popen()** процесса.

mkfifo() — создает именованный канал в рамках программы.

opendir() — открывает файл каталога UNIX (указатель позиции ввода-вывода в полученном дескрипторе установлен на первую запись каталога).

closedir() — закрывает файл каталога.

socket() — создает файл гнезда и в дальнейшем осуществлять обмен информацией через гнезда.

msgget() — создает новый или открывает уже существующий канал (очередь) сообщений.

mmap() — отображает открытый файл в оперативную память. Один из аргументов — пользовательский дескриптор открытого файла.

read() — считывает из открытого файла указанное количество байтов. Это единственная функция чтения, которая работает в системе без буферизации.

write() — записывает в файл указанное количество байтов. Это единственная функция записи, которая работает в системе без буферизации.

chmod() — изменяет права доступа к файлу.

remove() — удаляет указанный файл.

link() — создает жесткую ссылку на файл, т.е. присвоить файлу альтернативное имя.

unlink() — удаляет жесткую ссылку на файл или сам файл, если это — последняя жесткая ссылка на файл.

dup(), dup2() — позволяют скопировать пользовательский дескриптор открытого файла. Иногда это важно для обмена информацией между процессами.

symlink() — создает символьную (мягкую, символическую) ссылку на файл.

Утилиты, позволяющие управлять файлами:

touch — создает обычный файл в любом каталоге ФС.

cat [-u] [f1 f2 f3 ...] — выводит содержимое обычного файла на монитор (стандартный вывод). С помощью утилиты **cat** можно совершить конкатенацию (сцепление) файлов, т.е. объединить несколько файлов в один файл. **-u** — флаг для изменения размера выходного блока.

mv [-флаги] f1 f2 — перемещает/переписывает/переименовывает обычные файлы и каталоги.

cp [-флаги] f1 f2 — копирует обычные файлы и каталоги, позволяет скопировать один файл f1 в другой файл f2 или один/несколько файлов в каталог, если последний аргумент является именем существующего каталога. Если f2 существует, то его содержимое будет потеряно.

red имя (или ed имя) — вызов экранного редактора для создания или корректировки файла. См. [статью](#).

pr [-флаги] [f1 f2 ...] — форматирование обычного файла или файлов перед выводом на терминал, т.е. представить длинный файл или файлы в виде двух или трех колонок. См. [статью](#). Флаги:

-h — следующий за **h** текст трактуется как заголовок

-wn — задает ширину в **n** символов (вместо 72 по умолчанию)

-In — устанавливает длину страницы в **n** строк (вместо 66 по умолчанию)

-t — не печатать принятые по умолчанию 5 строк заголовка и 5 последних строк

-s! — колонки разделяются символом «!» вместо табуляции

-m — печатать все файлы одновременно, каждый в своей колонке

Ipr [-флаги] [f1 f2 ...] — отправляет обычный файл в очередь на печать. Позволяет печатать файлы одновременно с выполнением некоторых других. Файлы помещаются в спулинг и печатаются по мере освобождения печатающего устройства. **Ipr (line print)** — пользовательская команда печати. Программа **Ipr** принимает подлежащие печати данные и помещает их в спул, где их находит **LPD (Line Printer Daemon** — демон печати, реализующий одноименный протокол **Line Printer Daemon Protocol** — «протокол демона построчной печати») и выводит на печать. Программа **Ipr** — единственная программа, которая может ставить новые задания в очередь печати. Другие программы, которым необходимо использовать печать, обращаются для этого к **Ipr**. Флаги:

-r — удалить файл после печати.

-m — заказать почтовое сообщение об окончании печати файла.

-#N — количество копий печати (**N**).

-Pprinter — определяет какой принтер использовать.

du [-s][-a] имя — выводит размеры файлов в блоках и общее количество блоков для всех файлов.

По умолчанию блок 512 байт, но всегда можно указать, в каких блоках вы хотите получить размер файла.

Флаги:

-a — выдает размеры для каждого файла, а не только для каталогов.

-b — выдает размеры файлов в байтах.

-s — выдает только суммарный итог для каждого аргумента.

mkdir — создает каталоги, подкаталог в каком-то каталоге.

rmdir — удаляет пустые каталоги, которые не содержат никаких других файлов, кроме файла с именем «..» (ссылка на текущий каталог) и файла с именем «..» (ссылка на родительский каталог).

rm [-флаги] имя — удаляет файлы. Флаги:

-f — игнорировать несуществующие файлы и никогда не запрашивать подтверждение на удаления, т.е., по сути, это принудительное удаление файла. Однако этот параметр не удаляет файлы из каталога, если он защищен от записи.

-i — выдавать запрос на удаление каждого файла. Если ответ не утвержден, то файл пропускается.

-r — рекурсивно удалять содержимое каталогов.

chmod — изменяет права доступа к файлу.

chown — изменяет идентификатор владельца файла. Используется в случаях, когда вы хотите отдать свой файл коллегам, чтобы они его дальше дорабатывали.

chgrp — изменяет идентификатор группы владельца файла. Используется в случаях, когда вы хотите отдать свой файл коллегам, чтобы они его дальше дорабатывали.

cmp [-флаги] [f1 f2] — сравнивает содержимое двух обычных файлов. Возвращает 0 — в случае, если нет различий в файлах, 1 — если файлы различаются, 2 — если один из файлов не доступен. Выводит на экран первое несовпадение в сравниваемых файлах (номер записи и байт в записи, где произошло первое отличие).

См. также [статью](#). Флаги:

-s — не выводить текстовый вывод, генерирующийся по умолчанию.

-I — печатать позицию (номер отличающегося байта, десятичное смещение) и значения (восьмеричные) различающихся байтов в этой позиции из первого и второго файлов.

-n N — ограничить количество байтов, которые следует сравнивать, N байтами.

find [-флаги] — поиск файла в любых каталогах ФС UNIX с указанными параметрами. Имеет неограниченное количество внешних аргументов, т.е. характеристики поиска можно задавать очень большим количеством внешних аргументов. См. [статью](#). Флаги:

-name — искать по имени файла, при использовании подстановочных образцов параметр заключается в кавычки.

-type — тип искомого: **f**=файл, **d**=каталог, **l**=ссылка (*link*), **s**=гнездо(*socket*), **b**=специальный блок-ориентированный, **c**=специальный байт-ориентированный или **p**=канал.

-user — владелец: имя пользователя или *UID*.

-group — владелец: группа пользователя или *GID*.

-perm — указываются права доступа.

-size — размер: указывается в 512-байтных блоках или байтах (признак байтов — символ «c» за числом).

-atime — время последнего обращения к файлу в сутках назад.

- ctime** — время последнего изменения владельца или прав доступа к файлу в сутках назад.
- mtime** — время последнего изменения файла в сутках назад.
- newer** — искать файлы, созданные позже указанной даты.
- delete** — удалять найденные файлы.
- print** — показывает на экране найденные файлы. Это действие по умолчанию.
- exec** — используется для выполнения произвольных команд для каждого найденного файла.

Командная строка должна заканчиваться символом «;», который вы должныdezактивировать, чтобы shell его не интерпретировал; положение в файле отмечается при помощи {}.

ar — объединяет файлы в библиотеку или в архив. Удобно в случае, когда файлов очень много, сгруппировать их в библиотеку и затем пользоваться уже этой библиотекой.

file [-флаги] имя — определяет тип файла и анализирует его содержимое. Она анализирует первые записи файла и на основе этого анализа выводит свою характеристику указанного файла. Она может показать тип файла — текстовый (в какой кодировке и на каком языке программирования он написан) или двоичный (исполняемый файл или картинка и т.д.). Флаги:

- b** — не выводить имена файлов в вывод.
- f file** — считывает из указанного файла список файлов для проверки.
- F d** — указывает строку-разделитель d имени файла и его типа в выводе (разделитель по умолчанию — двоеточие).
- L** — определяет тип файлов, указанных по ссылке.
- z** — определяет тип файлов, находящихся в сжатых файлах.

wc [-флаги] [f1 f2 ...] — выводит на монитор количество записей (или строк), полей (слов) и байтов (символов) в указанном файле или файлах. Слова утилиты **wc** отбирают либо по пробелам, либо по табуляциям, либо по символам конца строки. Строки считаются по символам конца строки. Флаги:

- l** — вывести количество строк.
- c** — вывести количество байт.
- m** — вывести количество символов.
- w** — вывести количество слов.
- L** — вывести длину самой длинной строки.

more — постраничный вывод содержимого файла на монитор. Позволяет поэкранно печатать файл для удобства просмотра его содержимого. Как говорят студенты, утилита **more** «плется» экранами.

ls [-флаги] [имя_каталога] — выводит содержимое указанного каталога на монитор. Можно распечатать все файлы, которые в данный момент находятся в файловой системе. Флаги:

- l** — вывод полной информации о файле;
- a** — вывод всех файлов каталога, включая те файлы, имена которых начинаются с точки (скрытые);
- s** — вывод размера файла в блоках;
- d** — вывод информации только о подкаталогах (выдавать имена каталогов, как будто они обычные файлы, вместо того, чтобы показывать их содержимое);
- u** — сортирует список файлов по времени последнего доступа;
- t** — сортирует список файлов по времени последней модификации;
- i** — позволяет получить номер индексного дескриптора (*i-node*) вместо вида файла. Этот номер однозначно идентифицирует каждый файл в каждой файловой системе.

grep 'шаблон' <файл/файлы> [-флаги] — поиск данных в файле, удовлетворяющих заданному шаблону.

Выводит все записи (строки) файла, которые удовлетворяют указанному шаблону. См. также [эту](#) статью или [другую](#). Флаги:

- b** — перед строкой показывает номер блока, в котором она была найдена. Это может пригодиться при поиске блоков по контексту (блоки нумеруются с 0).
- c** — выдает только количество строк, в которых найдено соответствие с шаблоном.
- h** — предотвращает выдачу имени файла, содержащего сопоставившуюся строку, перед собственно строкой. Используется при поиске по нескольким файлам.
- i** — игнорирует регистр символов при сравнениях.
- I** — выдает только имена файлов, содержащих сопоставившиеся строки, по одному в строке. Если образец найден в нескольких строках файла, имя файла не повторяется.
- n** — выдает перед каждой строкой ее номер в файле (строки нумеруются с 1).
- s** — подавляет выдачу сообщений о не существующих или недоступных для чтения файлах.
- v** — выдает все строки, в которых не найден искомый шаблон.
- w** — поиск предполагает, что заданный шаблон должен быть целым отдельным словом, как если бы оно было окружено метасимволами |< и >|.

sort [-флаги] [f1 f2...] — сортировка или соединение файлов с помещением результата в заданный файл.

Флаги:

- b** — игнорировать пробелы в начале сортируемых полей или в начале ключей.
- d** — воспринимать в составе ключей лишь буквы латинского алфавита, цифры и пробелы, игнорируя все прочие символы.
- f** — игнорировать регистр букв.
- i** — в ключах рассматриваются только печатаемые (ASCII) символы, а остальные игнорируются.
- n** — сравнение строк ведётся по числовому значению (используют совместно с параметром -**b**).
- r** — сортировать в обратном порядке (по убыванию).
- o** f1 — выводит результат в указанный файл f1 вместо стандартного вывода.
- t** ch — использовать ch в качестве разделителя полей, а не переход от непробельных знаков к пробельным.
- z** — вместо символа новой строки, завершает строки двоичным 0 (NUL).

+ Отличие системных вызовов от утилит

Системные вызовы (СВ) применяются в программных кодах, когда вы разрабатываете программное приложение, а утилиты — это уже готовые программы, которые работают в командной строке и с помощью которых вы можете получить информацию о файле.

В отличии от системных вызовов, утилиты не имеют стандартов. Каждую утилиту каждый разработчик не сумел стандартизировать. Поэтому в разных системах одни и те же утилиты могут работать немного по-разному. Поэтому прежде чем использовать утилиту в своих программных приложениях, надо исследовать, какие возможности предоставляет каждая отдельная утилита от каждого отдельного разработчика. Утилиты так же называют командами, а по своей сути это готовые программные решения. Утилиты, идущие в рамках ОС — это программы, которые помогают пользователю смотреть за различными действиями, которые осуществляют ОС.

6. Системные вызовы работы с файлами в ОС UNIX

Автоматически открываемые файлы для любого процесса

Операции ввода-вывода в ОС UNIX связаны с вводом-выводом в файл. Файлы стандартного ввода-вывода и протокола ошибок автоматически открываются для любого вновь порожденного процесса и получают пользовательские дескрипторы файла 0,1,2 соответственно. При создании любой процесс открывает три файла. В ТПДОФ (таблице пользовательских дескрипторов открытых файлов процесса) появляются первые три записи (дескрипторы) об этих файлах. Эти дескрипторы ассоциируются с одним и тем же специальным файлом /dev/tty, соответствующим терминалу данного пользователя:

0. /dev/tty_r — пользовательский дескриптор файла стандартного ввода (ввод с клавиатуры).
1. /dev/tty_w — пользовательский дескриптор файла стандартного вывода (вывод на монитор).
2. /dev/tty_w — пользовательский дескриптор файла стандартного протокола (сообщения об ошибках выводятся на монитор).

Таблица пользовательских дескрипторов открытых файлов (ТПДОФ) вновь порожденного процесса:

| N | Имя файла | Ссылка на запись таблице файлов |
|---|-----------------------|---------------------------------|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| | | |
| | | |

Сама ТПДОФ процесса относится к контексту процесса, она не является объектом ядра, т.е. вместе с процессом она может подвергаться свопингу (выгрузке во внешнюю память). Также она пропадает вместе с завершением процесса.

Системный вызов close()

СВ close отсоединяет открытый файл от процесса, т.е. закрывает файл:

```
#include <unistd.h>
int close(int fd);
```

Возвращает 0 в случае успеха и -1 в случае неудачи.

Закрытие файла приводит также к снятию любых блокировок, которые могли быть наложены процессом. При завершении процесса все открытые им файлы автоматически закрываются ядром. Многие приложения используют это обстоятельство и не закрывают файлы явно.

Системный вызов open(). Примеры реализации

Для работы с файлами — их открытия и создания, используется СВ open(). Рассмотрим его синтаксис:

```
#include <fcntl.h> // Описание макросов для режимов открытия файлов
```

```
int open( char *name, int flag );
```

*name — указатель на строку символов, содержащую полное имя открываемого файла (указываются все подкаталоги, начиная с корневого). Если указывается неполное имя файла, то он будет искааться в текущем каталоге.

flag — режим открытия файла — это целые значения, указывающие, какие виды доступа к файлу разрешены. Основные из них:

0. Открыть для чтения (O_RDONLY).
1. Открыть для записи (O_WRONLY).
2. Открыть и для чтения, и для записи (O_RDWR).

Таблица флагов (flag)

| | <fcntl.h> |
|-----|-----------|
| 0 | O_RDONLY |
| 1 | O_WRONLY |
| 2 | O_RDWR |
| ... | ... |

Флаг (макрос) O_APPEND

По умолчанию при открытии файла указатель чтения/записи всегда устанавливается на первый байт файла, если не был установлен флаг (макрос) O_APPEND. Если файл открыт с флагом O_APPEND, то перед началом каждой операции записи файла указатель текущей позиции устанавливается в конец файла. По окончании записи значение текущей позиции увеличивается на количество фактически записанных байтов.

Можно не использовать заголовочный файл fcntl.h, просто указывая числовые значения.

Когда процесс вызывает СВ open() или creat() для того, чтобы открыть файл для чтения или записи, ядро ОС преобразует имя файла в индексный дескриптор файла. Если индексный дескриптор файла доступен процессу (есть право доступа), то ядро ищет в ТПДОФ процесса первую свободную строку. Номер этой строки будет возвращен процессу, как пользовательский дескриптор открытого файла. Итак, СВ open() возвращает:

- Дескриптор файла в случае успеха (целое неотрицательное число) — номер первой (наименьшей) свободной записи в ТПДОФ процесса (начиная от 0), в которую была записана информация об открытом файле.
- -1 — в случае ошибки, когда либо задано неправильное имя файла (файл не найден), либо права доступа не позволяют открыть этот файл в том режиме, который вы указали.

Примеры реализации системного вызова open()

Рассмотрим случай, когда в текущем каталоге существует файл a.txt, а права доступа допускают чтение и запись в этот файл.

```
#include <fcntl.h>
int fd; // Переменная для хранения пользовательского файлового дескриптора файла.
// Файл a.txt из текущего каталога открывается для чтения:
fd = open("a.txt", 0); // fd = 3
// Эквивалентная запись открытия файла с использованием макроса из файла fcntl.h:
// fd = open("a.txt", O_RDONLY);
ТПДОФ процесса:
```

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _r | ... |
| | | |
| | | |
| | | |

close(1); // Удаление записи с номером 1 из ТПДОФ процесса

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| | | |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _r | ... |
| | | |
| | | |
| | | |

// Файл a.txt из текущего каталога открывается для записи:

```
fd = open("a.txt", 1); // fd = 1
```

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | a.txt _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _r | ... |
| | | |
| | | |
| | | |

/* Файл a.txt из текущего каталога открывается для записи, указатель чтения/записи открытого файла перемещается на конец файла для того, чтобы можно было добавить информацию в конец файла: */

```
fd = open("a.txt", O_WRONLY | O_APPEND); // fd = 4
```

/* Знак «|» обозначает побитовое сложение режимов открытия файла. */

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | a.txt _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _r | ... |
| 4 | a.txt _w | ... |
| | | |
| | | |

Системный вызов creat(). Права доступа к файлу

Права доступа к файлу

Права доступа — важная особенность систем UNIX, которая позволяет защищать информацию, хранимую в файлах. У каждого файла есть следующие категории пользователей:

- **Владелец (user)** — владелец файла;
- **Группа (group)** — несколько пользователей, которым владелец разрешил пользоваться файлом;
- **Прочие (others)** — все остальные пользователи, которые могут работать с файлом.

При создании файла ОС первоначально присваивает какие-то права доступа к нему вне зависимости от того, с помощью какого средства создается этот файл. Права доступа всегда можно поменять в процессе работы с файлом.

Права доступа к файлу — это целое число. Изначально оно занимало 2 байта (16 бит), хотя сейчас под права доступа отводится 4 байта (32 бита), все равно рассматриваются последние 16 бит. Непосредственно за сами права доступа отвечают младшие 9 бит (с нулевого по восьмой), по три бита на каждую из категорий пользователей:

| владелец (user) | | | группа (group) | | | прочие (others) | | |
|-----------------|-----|-----|----------------|-----|-----|-----------------|-----|-----|
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| r | w | x | r | w | x | r | w | x |
| 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |

У каждого бита из тройки есть свое предназначение:

- **r** — read (чтение);
- **w** — write (запись);
- **x** — execute (выполнение).

Значение бита может быть либо **1** (разрешить), либо **0** (запретить).

| | |
|-----|---|
| 000 | 0 |
| 001 | 1 |
| 010 | 2 |
| 011 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |

Представление десятичных (и восьмеричных) чисел с помощью трех битов:

Поскольку при создании файла вы являетесь его владельцем, выставляйте для себя разрешающие права доступа, позволяющие в дальнейшем комфортно работать с этим файлом.

Рассмотрим пример того, как будут отображаться права доступа 0644. Когда перед числом в тексте программы стоит ноль, то это указание транслятору воспринимать это число в восьмеричной системе исчисления. Если перед числом ноль не указывать, то транслятор будет воспринимать это число в десятичной системе исчисления. Следовательно:

$$0644_8 = 420_{10} = 110\ 100\ 100_2$$

Как видно из двоичной записи последних 9 бит числа, владелец может читать и писать в этот файл, группа и прочие пользователи могут только читать этот файл.

| USER | GROUP | OTHERS |
|------|-------|---------------|
| 1 | 1 | 0 1 0 0 1 0 0 |

Системный вызов creat()

СВ **creat()** создает обычный файл с указанными именем и правами доступа и открывает его на запись:

```
#include <fcntl.h> // Описание макросов для режимов открытия файлов
int creat(char* name, mode_t mode);
```

***name** — указатель на строку символов, содержащую полное имя создаваемого файла (указываются все подкаталоги, начиная с корневого). Если указывается неполное имя файла, то файл будет создан в текущем каталоге.

mode — права доступа к файлу.

Если файл, имя которого определяется в **name**, существовал ранее, и права позволяют осуществлять в него запись, то содержимое этого файла будет потеряно.

СВ **creat()** возвращает:

- **Дескриптор файла** в случае успеха (целое неотрицательное число) — номер первой (наименьшей) свободной записи в ТПДОФ процесса (начиная от 0), в которую была записана информация об открытом файле, доступном только для записи.
- -1 — в случае ошибки, когда права доступа запрещают осуществлять запись в уже существующий файл.

СВ **creat()** является производным от СВ **open()** и равнозначен записи:

```
open(filename, O_CREAT | mode);
```

Знак «|» здесь обозначает побитовое сложение прав доступа. Вне зависимости от того, какие права доступа вы даете файлу, если создаваемый файл не существует, то СВ **creat()** всегда создаст файл с любыми правами доступа и откроет его на запись, даже если при создании этого файла вы не даете прав владельцу на запись в этот файл. Если файл с заданным именем уже существует, то, в зависимости от прав доступа, вы можете и не пересоздать этот файл. Т.е. если права доступа запрещают осуществлять запись в уже существующий файл, то СВ **creat()** не создаст этот файл заново и не откроет его на запись, а вернет -1.

Пример

Если изначально в текущей директории нет файлов a.txt, b.txt, c.txt и d.txt:

```
#include <fcntl.h>
int fd;
fd = creat("a.txt", 0653); // fd = 3
```

ТПДОФ процесса:

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _w | ... |

Права доступа создаваемого файла:

$$0653_8 = 427_{10} = 110\ 101\ 011_2$$

| владелец (user) | | | группа (group) | | | прочие (others) | | |
|-----------------|---|---|----------------|---|---|-----------------|---|---|
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| r | w | x | r | w | x | r | w | x |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |

fd = creat("a.txt", 66); // 102 fd = 4

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _w | ... |
| 4 | a.txt _w | ... |

Файл будет создан поверх того файла, который уже существовал, т.к. права доступа уже существующего файла разрешают осуществлять запись в него.

$66_{10} = 0102_8 = 001\ 000\ 010_2$

| владелец (user) | | | группа (group) | | | прочие (others) | | |
|-----------------|---|---|----------------|---|---|-----------------|---|---|
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| r | w | x | r | w | x | r | w | x |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

fd = creat("a.txt", 0600); // fd = -1

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _w | ... |
| 4 | a.txt _w | ... |

Файл a.txt не будет создан поверх того файла, который уже существовал, т.к. права доступа уже существующего файла запрещают осуществлять запись в него (владелец файла не имеет права на запись), поэтому СВ creat() вернет -1.

$0600_8 = 384_{10} = 110\ 000\ 000_2$

| владелец (user) | | | группа (group) | | | прочие (others) | | |
|-----------------|---|---|----------------|---|---|-----------------|---|---|
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| r | w | x | r | w | x | r | w | x |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

fd = creat("b.txt", 075); // fd = 5

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _w | ... |
| 4 | a.txt _w | ... |
| 5 | b.txt _w | ... |

Вне зависимости от того, какие права доступа были заданы файлу, если создаваемый файл не существует, то СВ creat() всегда создаст файл с любыми правами доступа и откроет его на запись, даже если при создании этого файла вы не даете прав владельцу на запись в этот файл.

$075_8 = 61_{10} = 000\ 111\ 101_2$

| владелец (user) | | | группа (group) | | | прочие (others) | | |
|-----------------|---|---|----------------|---|---|-----------------|---|---|
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| r | w | x | r | w | x | r | w | x |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |

```
fd = creat("b.txt", 0644 ); // fd = -1
```

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _w | ... |
| 4 | a.txt _w | ... |
| 5 | b.txt _w | ... |

Файл b.txt не будет создан поверх того файла, который уже существовал, т.к. права доступа уже существующего файла запрещают осуществлять в него запись (владелец файла не имеет права на запись), поэтому СВ **creat()** вернет -1.

| владелец (user) | | | группа (group) | | | прочие (others) | | |
|-----------------|---|---|----------------|---|---|-----------------|---|---|
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| r | w | x | r | w | x | r | w | x |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |

```
fd = creat("c.txt", 6); // fd = 6
```

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _w | ... |
| 4 | a.txt _w | ... |
| 5 | b.txt _w | ... |
| 6 | c.txt _w | ... |

Вне зависимости от того, какие права доступа были заданы файлу, если создаваемый файл не существует, то СВ **creat()** всегда создаст файл с любыми правами доступа и откроет его на запись, даже если при создании этого файла вы не даете прав владельцу на запись в этот файл.

$$6_{10} = 006_8 = 000\ 000\ 110_2$$

| владелец (user) | | | группа (group) | | | прочие (others) | | |
|-----------------|---|---|----------------|---|---|-----------------|---|---|
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| r | w | x | r | w | x | r | w | x |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

СВ **open()**, равнозначный первому СВ **creat()**:

```
fd = open("d.txt", O_CREAT | 0653); // fd = 7
```

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | a.txt _w | ... |
| 4 | a.txt _w | ... |
| 5 | b.txt _w | ... |
| 6 | c.txt _w | ... |
| 7 | d.txt _w | ... |

0653₈ = 427₁₀ = 110 101 011₂

| владелец (user) | | | группа (group) | | | прочие (others) | | |
|-----------------|---|---|----------------|---|---|-----------------|---|---|
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| r | w | x | r | w | x | r | w | x |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |

Системные вызовы dup(), dup2(). Примеры реализаций

Копирование пользовательских дескрипторов открытых файлов

СВ **dup()** и **dup2()** позволяют скопировать пользовательский дескриптор открытого файла процесса. Иными словами, СВ **dup()** и **dup2()** позволяют осуществить доступ к одному и тому же файлу из одного и того же процесса через два различных пользовательских дескриптора файла. Это может быть полезно для обмена информацией между процессами.

Системный вызов dup()

int dup(int fd);

fd — номер копируемого пользовательского дескриптора открытого файла, т.е. номер записи в ТПДОФ (таблице пользовательских дескрипторов открытых файлов) процесса.

Возвращает:

- **Новый дескриптор файла**, т.е. порядковый номер первой (наименьшей) свободной записи в ТПДОФ (таблице пользовательских дескрипторов открытых файлов) процесса (начиная от 0), в которую была записана копия дескриптора.
- **-1** — в случае ошибки (например, если под номером копируемого дескриптора запись отсутствует, т.е. копировать нечего).

Пример

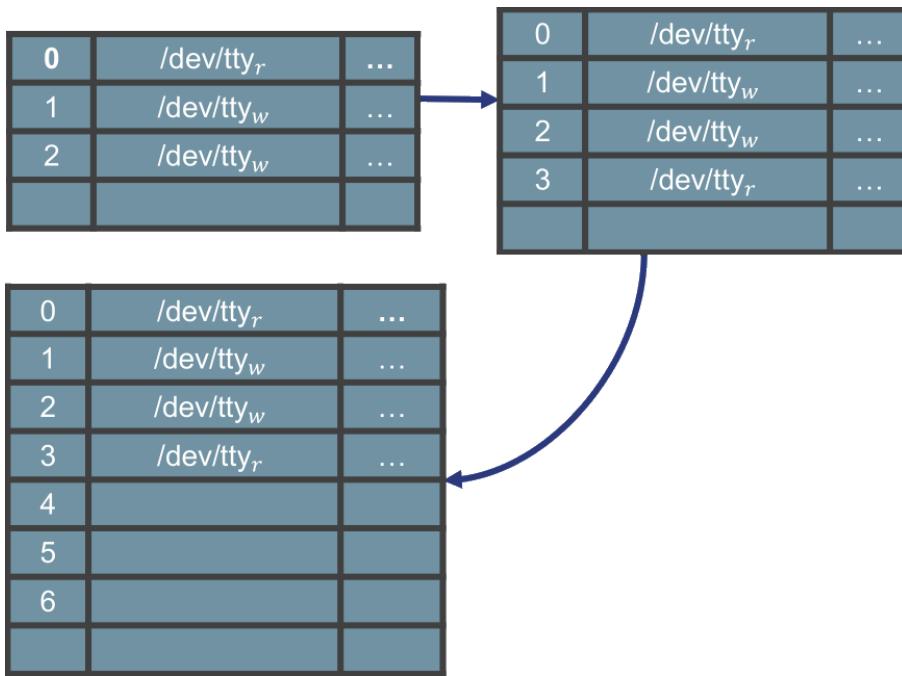
```
int fd;
fd = dup(0); // fd = 3
```

ТПДОФ процесса:

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | /dev/tty _r | ... |

fd = dup(6); // fd = -1 (запись под номером 6 отсутствует)



Системный вызов dup()

```
int dup2(int old_fd, int new_fd);
```

old_fd — номер копируемого пользовательского дескриптора открытого файла, т.е. номер записи в ТПДОФ (таблице пользовательских дескрипторов открытых файлов) процесса.

new_fd — номер дескриптора (номер записи в ТПДОФ), в который нужно осуществить копирование (целевой дескриптор).

СВ **dup2()** копирует дескриптор, предварительно закрывая целевой дескриптор, если он уже открыт данным процессом. Возвращает:

- **Переданный new_fd.** Если такой дескриптор уже открыт данным процессом, то **dup2()** закроет этот файл.
- **-1** — в случае ошибки (например, если под номером копируемого дескриптора запись отсутствует, т.е. копировать нечего).

Пример

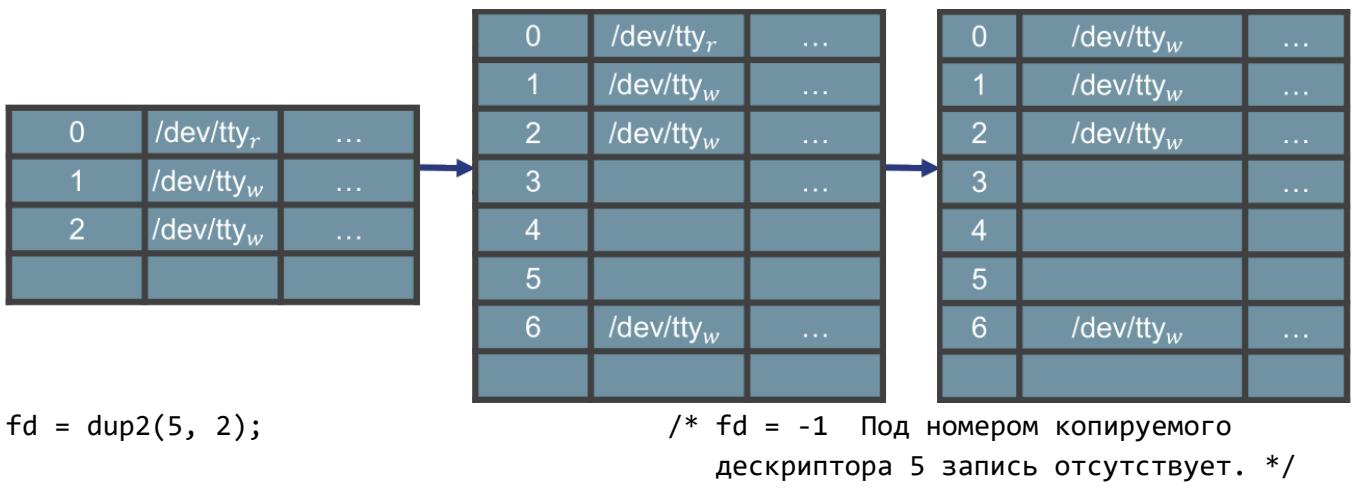
```
int fd;
fd = dup2(1, 6); // fd = 6
ТПДОФ процесса:
```

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | | ... |
| 4 | | |
| 5 | | |
| 6 | /dev/tty _w | ... |

```
fd = dup2(2, 0);
```

/* fd = 0 СВ **dup2()** копирует дескриптор, предварительно закрывая целевой дескриптор 0. */



Перенаправление стандартного ввода-вывода

С помощью СВ из группы **dup** можно осуществлять перенаправление стандартного ввода-вывода.

Как уже было сказано выше, операции ввода-вывода в ОС UNIX связаны с вводом-выводом в файл. Файлы стандартного ввода-вывода и протокола ошибок автоматически открываются для любого вновь порожденного процесса и получают пользовательские дескрипторы файла 0,1,2 соответственно. При создании любой процесс открывает три файла. В ТПДОФ (таблице пользовательских дескрипторов открытых файлов процесса) появляются первые три записи (дескрипторы) об этих файлах. Эти дескрипторы ассоциируются с одним и тем же специальным файлами /dev/tty, соответствующим терминалу данного пользователя:

0. /dev/tty_r — пользовательский дескриптор файла стандартного ввода (ввод с клавиатуры).
1. /dev/tty_w — пользовательский дескриптор файла стандартного вывода (вывод на монитор).
2. /dev/tty_w — пользовательский дескриптор файла стандартного протокола (сообщения об ошибках выводятся на монитор).

По умолчанию все эти дескрипторы ассоциированы с терминалом. То есть любая программа, не использующая другие дескрипторы, кроме стандартных, будет ожидать ввода с клавиатуры терминала. Весь вывод этой программы, включая сообщения об ошибках, будет происходить на экран терминала. Для таких программ можно осуществлять перенаправление ввода-вывода. Например, можно подавить вывод сообщений об ошибках, установить ввод или вывод из файла и даже передать вывод одной программы на ввод другой, используя СВ **fork()**.

Керриск M. *Linux API. Исчерпывающее руководство*. — СПб.: Питер, 2019 г., стр. 587:

*Все файловые дескрипторы, открываемые программой, которая вызывает **exec()**, остаются открытыми на протяжении выполнения этого вызова и доступны для использования в новой программе. Часто это может быть полезным, потому что файлы, открытые вызывающей программой в определенных дескрипторах, автоматически становятся доступными для новой программы (которая при этом не должна знать их имена или открывать их заново).*

Пример реализации

В следующем примере в текущем процессе формируется процесс-сын, который осуществляет выполнение утилиты **ls** с ключем **-a**, при этом перенаправляя вывод результатов этой утилиты в файл **a.txt**, которого до запуска программы не существует.

lect64-1.c:

```

#include <stdio.h> // SYSTEM V POSIX.1
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
void main()

```

```
{
// Выделение памяти под целочисленные переменные:
int fd, s, d;
/* Процесс при помощи СВ fork() разделяется на две идентичные копии,
которые начинают выполняться параллельно. */
if (fork() != 0)
{ /* Тело процесса-отца
ТПДОФ процесса-отца останется неизменной:


|   |                       |     |
|---|-----------------------|-----|
| 0 | /dev/tty <sub>r</sub> | ... |
| 1 | /dev/tty <sub>w</sub> | ... |
| 2 | /dev/tty <sub>w</sub> | ... |
|   |                       |     |


*/
/* Процесс-отец выполняет СВ wait() и ждет завершения выполнения
процесса-сына. */
d = wait(&s);
printf("s = %d\n", s);
/*
Дождавшись, когда процесс-сын завершится после выполнения СВ exec1(),
процесс-отец может прочитать информацию из файла и проанализировать ее.
СВ wait() возвращает идентификатор завершившегося процесса-сына и в свой
единственный аргумент записывает причину гибели процесса-сына.
т.к. процесс-сын завершается с помощью СВ exec1(), который, в случае
удачного выполнения завершится с нулевым аргументом, то s = 0. Однако,
если произойдет ошибка, то процесс-сын дойдет до СВ _exit(2) и s = 512,
т.к. аргумент СВ _exit() записывается с 8 по 15 биты (биты с 0 по 7
заполняются тогда, когда процесс убивается сигналом и в них записываются
номер сигнала и действия процесса по сигналу). Следовательно, двойка,
записанная в двоичной системе исчисления в 9 и 8 биты:
s = 00000010 0 0000000 (2) = 512 (10)
s = 51210

| Аргумент exit() |    |    |    |    |    |   |   | Номер сигнала |   |   |   |   |   |   |   |
|-----------------|----|----|----|----|----|---|---|---------------|---|---|---|---|---|---|---|
| 15              | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7             | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0               | 0  | 0  | 0  | 0  | 0  | 1 | 0 | 0             | 0 | 0 | 0 | 0 | 0 | 0 | 0 |


*/
}
else
{ /* Тело процесса-сына */
/* В текущем каталоге создается файл a.txt с правами доступа 0664:
110 110 100, при этом в ТПДОФ процесса-сына появляется соответствующая
запись с номером 3, т.к. это порядковый номер первой (наименьшей)
свободной записи в ТПДОФ.

Права доступа к файлу a.txt:
USER GROUP OTHERS


|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|


*/
}
```

```

fd = creat("a.txt", 0664);           // fd = 3
/* Закрывается файловый дескриптор файла стандартного вывода, т.е.
   освобождается запись с номером 1 в ТПДОФ процесса-сына. */
close(1);
/* Запись с номером 1 теперь свободна. В нее копируется пользовательский
   дескриптор с номером fd. Теперь в ТПДОФ процесса-сына две одинаковые
   записи под номерами fd = 3 и 1. */
dup2(fd, 1);
/* В ТПДОФ процесса-сына освобождается запись с номером fd = 3. Теперь
   в ТПДОФ процесса-сына всего три записи с номерами 0, 1 и 2. Причем
   в записи под номером 1 вместо файла стандартного вывода (монитора) указан
   обычный файл a.txt, который открыт также на запись. Так произошло
   перенаправление стандартного вывода. */
close(fd);
/* ТПДОФ процесса-сына:



|   |           |     |
|---|-----------|-----|
| 0 | /dev/ttys | ... |
| 1 | /dev/ttw  | ... |
| 2 | /dev/ttw  | ... |
|   |           |     |



|   |           |     |
|---|-----------|-----|
| 0 | /dev/ttys | ... |
| 1 | /dev/ttw  | ... |
| 2 | /dev/ttw  | ... |
| 3 | a.txtw    | ... |
|   |           |     |



|   |           |     |
|---|-----------|-----|
| 0 | /dev/ttys | ... |
| 1 |           | ... |
| 2 | /dev/ttw  | ... |
| 3 | a.txtw    | ... |
|   |           |     |



|   |           |     |
|---|-----------|-----|
| 0 | /dev/ttys | ... |
| 1 | a.txtw    | ... |
| 2 | /dev/ttw  | ... |
| 3 | a.txtw    | ... |
|   |           |     |


*/
/* Выполнение утилиты ls с ключем -а, которая выводит все файлы текущего
   каталога, включая те файлы, имена которых начинаются с точки (скрытые).
   Результат выполнения этой инструкции вместо монитора будет выведен
   в файл a.txt: */
exec1("/bin/ls", "ls", "-a", 0);
/* Стивен Р.У., Раго С.А. UNIX. Профессиональное программирование.
   3-е издание. - СПб.: Питер, 2018 г., стр. 308-309
   Функция fork часто используется для создания нового процесса, который
   затем запускает другую программу с помощью одной из функций семейства
   execs. Когда процесс вызывает одну из функций execs, он полностью
   замещается другой программой, и эта новая программа начинает выполнение
   собственной функции main. Идентификатор процесса при этом не изменяется,
   поскольку функция execs не создает новый процесс, она просто замещает
   текущий процесс - его сегмент кода, сегмент данных, динамическую область
   памяти и сегмент стека - другой программой. */
/* Т.к. командная строка у СВ exec1() сформирована корректно, то до нижней
   Инструкции _exit(2) процесс уже не дойдет. Процесс-сын после выполнения

```

```
СВ exec1() будет успешно завершен с помощью СВ _exit(0); код 0 означает
правильное завершение программы. */
_exit(2);
/* Программа закрывается с помощью СВ _exit(); код 2 означает завершение
работы программы, вызванное ошибкой СВ exec1(). */
}
```

7. Управление устройствами ввода-вывода (УВВ) в ОС UNIX. Системные вызовы для ввода-вывода информации

Поддержка операций ввода-вывода — одна из важных функций ядра ОС.

В UNIX-подобных ОС появилось революционное нововведение, заключающееся в том, что все ресурсы, все атрибуты, которыми манипулирует ОС, унифицированы с понятием файл. Это относится и к системе управления УВВ.

Как уже говорилось ранее, специальные файлы предназначены для организации взаимодействия с УВВ (устройствами ввода-вывода). С каждым УВВ (накопитель на магнитном диске, клавиатура, монитор, мышь, принтер и т.д.) связан свой специальный файл. Бывают **байт-ориентированными и блок-ориентированными**. С ними можно работать как с обычными файлами, т.е. прикладная программа может выполнять операции чтения и записи со специальным файлом точно так же, как и с обычным файлом. ОС будет автоматически вызывать соответствующий драйвер устройства для выполнения фактической передачи данных между физическим устройством и данной программой. Т.е. с точки зрения пользователя работать с УВВ становится очень просто: открыли файл и записали/прочитали информацию.

7.1 Логическая организация системы управления УВВ

Итак, суть логической организации системы управления вводом-выводом — все УВВ есть (специальные) файлы и чтение и запись на них есть чтение и запись в файл. В ОС UNIX логическая организация ввода-вывода упрощена за счет наличия одинакового интерфейса ввода-вывода на уровне ядра системы.

С точки зрения логической организации, все УВВ представляются в виде файлов, все файлы рассматриваются как последовательный набор байтов, к которым возможно как последовательное, так и прямое обращение. Чтобы работать с таким файлом, его нужно открыть (для записи либо для чтения) с помощью СВ `open()`, с помощью СВ `Iseek()` возможен прямой доступ к любой позиции в файле и СВ `close()` позволяет закрыть этот файл и освободиться от устройства. Напомним, что при завершении процесса открытые файлы автоматически закрываются.

Например, нужно вывести информацию на принтер. Для этого достаточно открыть в режиме записи специальный файл, который соответствует принтеру, и записывать информацию в этот файл. Эта информация автоматически будет передаваться на устройство принтера.

7.1-7.2 Физическая организация системы управления УВВ

Физическая организация опирается на наличие программных средств, работающих на уровне ядра ОС — драйверов. Драйвер УВВ — программа обмена данными между конкретным УВВ и ОП компьютера, учитывающая все особенности данного УВВ. Ядро ОС устанавливает однозначную связь между драйверами и соответствующими специальными файлами с помощью таблиц. Поэтому при обращении к специальному файлу будет вызван соответствующий драйвер устройства.

Интерфейсы устройств ввода-вывода данных

Драйвер УВВ и система буферизации ядра ОС поддерживают два вида интерфейсов:

- **Блок-ориентированные.**
- **Байт- ориентированные.**

Блок-ориентированный интерфейс

Блок-ориентированный интерфейс позволяет осуществлять обмен блоками между устройством и ОП. Он дает возможность значительно повысить эффективность системы управления вводом-выводом за счет организации кэш-памяти. Кэш-память — системные буферы ввода-вывода с размером, кратным размеру блока, где оседают те блоки данных, обращение к которым производится наиболее часто. Когда возникает запрос на считывание какого-то блока данных с УВВ, то сначала просматривается кэш-память, и только в том случае, если необходимый блок там отсутствует, обычно производится выталкивание из кэш-памяти какого-то

блока и обращение через драйвер к внешнему устройству, с которого информация переносится в освобожденный блок кэш-памяти, откуда уже производится ее передача в ОП.

Часто такие устройства называют устройствами прямого доступа к памяти, или Direct Memory Access (DMA).

Байт-ориентированный интерфейс

Байт-ориентированный интерфейс предназначен в основном для медленных периферийных устройств.

Обмен данными осуществляется посимвольно через буфер символов в ядре ОС. При обмене данными (группами символов) без буферизации байт-ориентированный интерфейс называется **прозрачным блок-ориентированным**.

Для большей части внешних устройств допускается использование обоих видов (как блок- так и байт-ориентированных) интерфейсов.

Программные и табличные составляющие системы управления вводом-выводом

Система управления вводом-выводом (СУВВ) ОС UNIX — часть ядра, выполняющая функции диспетчера УВВ.

Программная часть диспетчера включает:

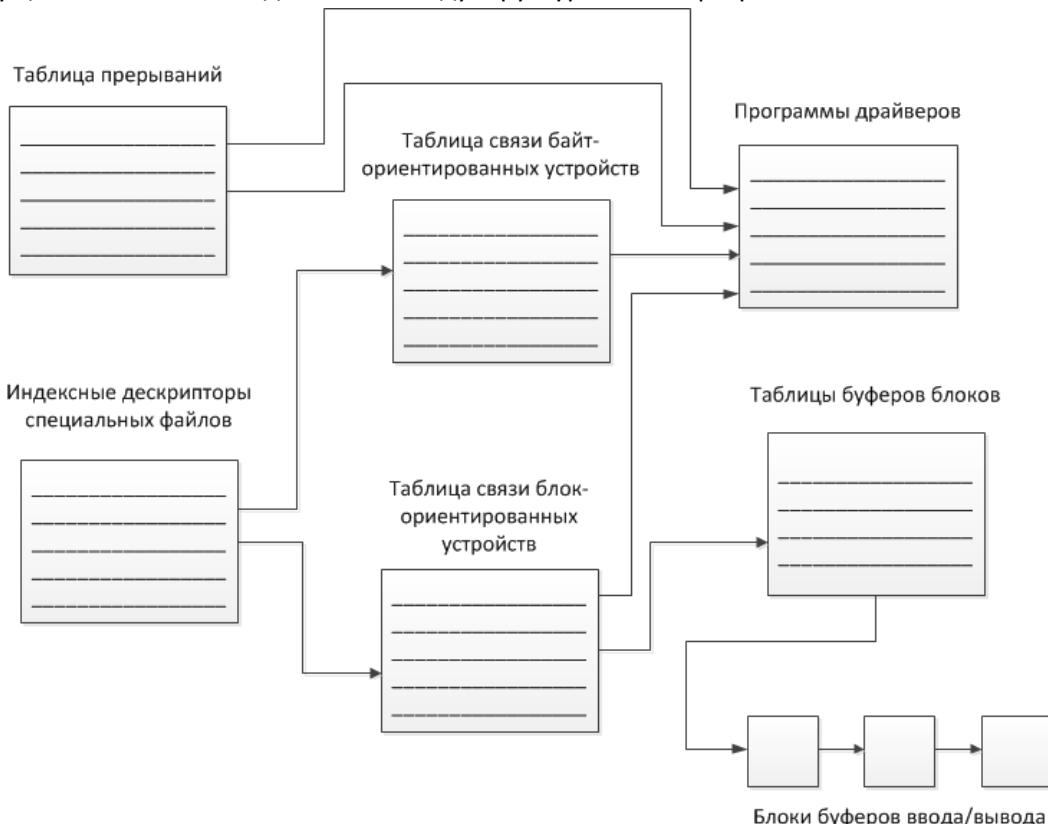
- Управляющую компоненту.
- Систему буферизации.
- Набор драйверов.

В качестве **табличных компонент** используются:

- Индексные дескрипторы специальных файлов.
- Таблица связи между драйверами байт-ориентированных устройств и ядром системы.
- Таблица связи между драйверами блок-ориентированных устройств и ядром системы.
- Таблица буферов блоков (кэш-память).
- Таблица векторов прерываний.

Таблицы системы управления устройствами ввода-вывода

Взаимодействие этих компонент можно отобразить в виде следующей схемы (стрелочками показано, как осуществляется взаимодействие между структурными и программными компонентами СУВВ):



Индексный дескриптор специального файла

Каждый индексный дескриптор связан с таблицами связи.

Индексный дескриптор специальных файлов включает информацию:

1. **Имя специального файла** (на специальный файл нельзя сделать ни жесткую ни символьную ссылку, т.е. этот файл будет в единичном экземпляре в рамках всей ОС) — **только для специальных файлов**. Имя файла не включается в индексный дескриптор обычного файла.
2. **Тип устройства** (блок (**b**)/байт (**c**)-ориентированное).
3. **Номер типа устройства (старший номер)** — характеризующий это устройство номер записи в таблице драйверов, которую ведет ОС (таблица ядра содержит список всех драйверов, известных системе).
4. **Номер устройства (младший номер)**. Этот номер сообщает драйверу, с каким конкретно физическим устройством он взаимодействует. Например, если в системе два абсолютно одинаковых диска, то старший номер (тип устройства) у них будет совпадать, а различаться будут младшие номера (номера устройств). Иначе говоря, этот номер используется для того, чтобы отличить одно одинаковое устройство (одного типа) от другого.

Таблицы связи между драйверами для блок-ориентированных и байт-ориентированных устройств

Каждый индексный дескриптор специального файла ссылается на соответствующую запись из таблиц связь. Каждая запись в этих таблицах относится к одному УВВ и адресуется к определенному драйверу, обеспечивающему обмен информацией между основной памятью и устройством. Таблица связи для блок-ориентированных устройств, кроме того, связана с таблицей буферов блоков, указывающей на буферы ввода-вывода, образующие кэш-память.

Эти две таблицы связи включены в ядро ОС и позволяют ядру быстро перестраиваться на новый состав и параметры устройств путем включения и отключения новых драйверов. В целом они предназначены для однозначного выбора программ драйверов (адресации к определенному драйверу).

Программы драйверов

Программы драйверов предназначены для открытия и закрытия специальных файлов, чтения и записи данных (передачи данных между устройством и специальным файлом), а также для управления режимами работы устройств. Каждый драйвер разделен на несколько частей:

- **Верхняя часть** драйвера работает в режиме вызывающего процесса и служит интерфейсом с остальной частью ОС. Это — видимая часть драйвера.
- **Нижняя часть** работает в контексте ядра и непосредственно взаимодействует с устройством, занимаясь передачей информации с УВВ в ОП.

Таблица прерываний

В таблице прерываний находятся векторы прерываний, т.е. адреса обработчиков прерываний от разных УВВ (каждая запись этой таблицы содержит адрес обработчика прерываний для каждого конкретного УВВ). Таблица прерываний обычно располагается в начальной области памяти (в верхней части ядра ОС) по фиксированным адресам. При возникновении прерываний от УВВ управление передается программе-обработчику прерываний драйвера, адрес которой выбирается из таблицы прерываний.

Пример реализации операции чтения с блок-ориентированного устройства ввода-вывода

Открыть файл, с которого собираемся читать (в режиме O_RDONLY = 0), с помощью СВ `open()`:

Если УВВ — клавиатура, то открываем файл устройства терминала

- `fd = open("dev/tty", 0);`

Если УВВ — диск, то

- `fd = open("dev/hdr0/home/lab/a.txt", 0);`

Для диска имя специального файла используется по умолчанию, поэтому его имя можно не указывать

- `fd = open("a.txt", 0);` // a.txt будет искааться в текущем каталоге.

Для байт-ориентированного УВВ операция чтения происходит проще, т.к. там используется буфер символов в ядре ОС, а для блок-ориентированных устройств используется кэш-память.

Пример реализации операции чтения с блок-ориентированного устройства ввода-вывода (магнитного диска)

Считать информацию с устройства hd0 с помощью СВ `read()`:

```
fd=open("/dev/hd0", 0);
read (fd, buf, n);
```

Инициализирует обращение к специальному файлу /dev/hd0 и через индексный дескриптор специального файла fd определяет тип и номер устройства, с которого необходимо прочитать информацию. Т.к. индексный дескриптор специального файла ссылается на соответствующую запись из таблиц связы (в данном случае из таблицы связи для блок-ориентированных устройств), то по адресу из таблицы связи управление передается необходимому драйверу.

Программа драйвера проверяет, не располагается ли требуемый блок информации в кэш-памяти:

- Если этот блок информации располагается в кэш-памяти, драйвер эмулирует сигнал прерывания об окончании обмена.
- В противном случае драйвер освобождает, когда это требуется, участок кэш-памяти (т.е. выталкивает из нее один блок, если в кэше нет свободного места) и запускает физическую операцию обмена (находит на диске этот блок данных, перемещает блок головок на нужную дорожку и в нужный сектор диска, затем считывает этот блок данных с УВВ и переносит этот блок данных в кэш-память). По окончанию операции обмена моделируется прерывание и запускается программа обработки прерываний драйвера, адрес которой извлекается из записи в таблице прерываний.

Далее драйвер переписывает информацию из кэш-памяти в область основной памяти, указанную программистом для ввода (переменная `buf`).

По окончании операции ввода управление обычно передается диспетчеру процессов ядра ОС UNIX.

Как видно из объяснения, даже при простой операции ввода, использовались все программные компоненты и табличные компоненты СУВВ.

7.3 Системные вызовы для ввода-вывода информации

Самый низкий уровень ввода-вывода в ОС UNIX представлен СВ `read()` и `write()`. Они не предусматривают ни какой-либо буферизации, ни какого-либо сервиса, а являются обращением непосредственно к ОС. Все остальные библиотечные функции, которые используются в ОС (`scanf()`, `printf()`, `fscanf()`, `fprintf()`, `fgets()`, `fputs()`, `getc()`, `putc()`, `fgetc()`, `fputc()`), работают с буферизацией. Т.е. СВ `read()` и `write()` — единственные, которые не используют буферизацию. По сравнению с другими СВ они самые медленные, поэтому их разрешено прерывать, в таком случае они возвращают `-1`.

Стандартные функции ввода-вывода `fgets()`, `fputs()`, `getc()`, `putc()`, `fgetc()`, `fputc()` и их эффективность рассмотрены в книге Стивен Р.У., Раго С.А. UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018 г., стр. 201-207.

Системный вызов `read()`

СВ `read()` выбирает блок данных фиксированного размера из файла с указанным пользовательским дескриптором `fd` и записывает его в буфер `buf`, расположенный в ОП компьютера:

```
#include <stdio.h>
int read(int fd, char* buf, int n);
```

Обеспечивает самый низкий уровень ввода в ОС. Не предусматривает буферизации, в отличии от всех библиотечных функций, которые используются в ОС. Является непосредственным обращением к ОС, считывая данные напрямую с конкретного УВВ.

fd — номер пользовательского дескриптора открытого для чтения файла (номер записи из ТПДОФ процесса),

buf — указатель на буфер в программе пользователя, куда должны поступить данные из файла,

n — количество байтов, которое требуется прочитать из файла с дескриптором **fd**.

В случае успешного завершения **возвращает количество байт, успешно прочитанных и сохраненных в аргументе buf**.

Обычно это число равно **n**. Но если в файле меньше **n** байт, то вернет число меньше **n**. Если после того, как все данные будут прочитаны, встретится признак конца файла, то вернет **0**. При ошибке чтения вернет **-1**, если файл с пользовательским дескриптором **fd** не удалось открыть или если во время операции ввода произошло прерывание.

Системный вызов write()

СВ **write()** помещает блок данных фиксированного размера из буфера **buf** (в ОП компьютера) в файл с пользовательским дескриптором **fd**:

```
#include <stdio.h>
int write(int fd, char* buf, int n);
```

Обеспечивает самый низкий уровень вывода в ОС. Не предусматривает буферизации, в отличии от всех библиотечных функций, которые используются в ОС. Является непосредственным обращением к ОС, записывая данные напрямую на конкретное УВВ.

fd — номер пользовательского дескриптора открытого для записи файла (номер записи из ТПДОФ процесса),

buf — указатель на буфер в программе пользователя, откуда берутся (считываются) данные для последующей их записи в файл,

n — количество байтов, которое требуется считать из **buf** и записать в файл с дескриптором **fd**.

В случае успешного завершения **возвращает количество байт, успешно записанных** в файл. Обычно это число равно **n**. Или **-1** в случае неудачи, если диск файловой системы переполнился или размер файла превышает критическое значение, а также если во время операции вывода произошло прерывание.

7.4 Примеры реализаций системных вызовов ввода-вывода информации

Пример 1.1

Пусть на диске в текущем каталоге находится файл a.txt с правами доступа 0600 (владелец может читать и писать в этот файл), в котором содержится запись: «Good morning my friend» (22 символа).

lect74-1.1.c:

```
// Заголовочные файлы:
#include <string.h> /* Содержит прототипы функций работы со строками string(),
                     strcasecmp(), strcat(), strcpy(), memset(), bzero() и т.д.
                     */
#include <fcntl.h> // Описание макросов для режимов открытия файлов
#include <unistd.h> /* Для возможности мобильного переноса с одной вычислительной
                     платформы на другую */
#include <stdio.h> /* Для поддержания стандартного ввода-вывода. Включает функции
                     scanf(), printf(), fgets(), fputs() и др. Функции stdio
                     наслаждаются поверх СВ ввода-вывода (open(), close(),
                     read(), write() и т.д.). */
```

```
void main()
```

```
{
    // Выделение памяти под переменные:
    // Символьный массив-буфер
    char buf[80];
    // Переменная для сохранения кода возврата СВ read()
    int result;
    /* Поскольку могут быть предоставлены грязные страницы памяти, то прежде чем
       использовать выделенную память, желательно ее очистить с помощью функции
       memset() или bzero(), занося нулевые значения в область памяти: */
    memset(buf, 80, 0);
    /* Процесс открывает для чтения файл a.txt, номер пользовательского дескриптора
       файла сохраняется в переменную fd: */
    int fd = open("a.txt", 0);           // fd = 3
    /* По умолчанию при открытии файла указатель чтения/записи всегда устанавливается
       на первый (с нулевым индексом) байт файла. */

    /* СВ read() читает цепочку заданной длины (50 байтов) из любых символов: */
    result = read(fd, &buf[0], 50); /* эквивалентно read(fd, buf, 50); = 22,
                                    т.е. сколько байтов было в файле. После
                                    выполнения СВ read() указатель чтения/записи
                                    отсчитал 22 байта от начала файла и уткнулся
                                    в конец файла. */
    result = read(fd, &buf[23], 50); /* = 0. При начале попытки чтения достигнут
                                    конец файла. Указатель чтения/записи все так
                                    же установлен на 22 байт от начала файла. */
}
}
```

Пример 1.2

Условия и заголовочные файлы те же, что и в примере 1.1.

lect74-1.2.c:

```
void main()
{
    // Выделение памяти под переменные:
    // Символьный массив-буфер
    char buf[80];
    // Переменная для сохранения кода возврата СВ read()
    int result;
    /* Поскольку могут быть предоставлены грязные страницы памяти, то прежде чем
       использовать выделенную память, желательно ее очистить с помощью функции
       memset() или bzero(), занося нулевые значения в область памяти: */
    memset(buf, 80, 0);
    /* Процесс открывает для чтения файл a.txt, номер пользовательского дескриптора
       файла сохраняется в переменную fd: */
    int fd = open("a.txt", 0);           // fd = 3
    /* По умолчанию при открытии файла указатель чтения/записи всегда устанавливается
       на первый (с нулевым индексом) байт файла. */

    /* СВ read() читает цепочку заданной длины (11 байтов) из любых символов: */
    result = read(fd, buf, 11);        /* = 11. После выполнения СВ read() указатель
                                    чтения/записи отсчитал 11 байт от начала
```

```

        файла. */
result = read(fd, &buf[12], 8); /* = 8. Указатель чтения/записи отсчитал 8 байт
                                и установился на 19 байт от начала файла. */
result = read(fd, &buf[20], 32); /* = 3. Указатель чтения/записи отсчитал 3
байта,
                                и установился на 22 байт от начала файла,
                                уткнувшись в конец файла. */
result = read(fd, &buf[23], 1); /* = 0. При начале попытки чтения достигнут
конец файла. Указатель чтения/записи все так
же установлен на 22 байт от начала файла. */
}

```

Пример 2

Пусть файл b.txt с правами доступа 0600 (владелец может читать и писать в этот файл) существует в текущем каталоге.

lect74-2.c:

Заголовочные файлы те же, что и в примере 1.1.

```

void main()
{
    // Выделение памяти под переменные:
    // Переменная для сохранения кода возврата CB write()
    int result;
    /* Процесс открывает для записи файл b.txt, номер пользовательского дескриптора
       файла сохраняется в переменную fd: */
    int fd = open("b.txt", 1);           // fd = 3
    /* По умолчанию при открытии файла указатель чтения/записи всегда устанавливается
       на первый (с нулевым индексом) байт файла. */
    printf("Файл открыт для записи, позиция указателя %ld \n", lseek(fd, 0,
SEEK_CUR));

    /* CB write() записывает в файл цепочку заданной длины (2 байта) из символьного
       массива "Hello": */
    result = write(fd, "Hello", 2); /* = 2. После выполнения CB write() указатель
                                    чтения/записи отсчитал 2 байта от начала
                                    файла. В файл запишутся символы "He". Если
                                    файл b.txt не пустой, то запись будет
                                    производиться поверх существующих данных. */
    result = write(fd, "Hello", 13); /* = 13. Указатель чтения/записи отсчитал
                                    13 байт и установился на 15 байт от начала
                                    файла. В файл запишутся 5 символов "Hello",
                                    а остальные 13-5=8 символов – это различный
                                    мусор, который находится в памяти. Это
                                    связано с тем, что в CB write() фактически
                                    передается указатель на начало буфера
                                    в программе пользователя, откуда берутся
                                    (считываются) данные для последующей их
                                    записи в файл. */

    /* В результате первые 7 символов файла b.txt будут представлять из себя
       последовательность "HeHello", а размер файла будет не менее 15 байт. */
}

```

Пример 3

Как было сказано выше, СВ **read()** и **write()** представляют самый низкий уровень ввода-вывода в ОС UNIX. Они не предусматривают никакой-либо буферизации, и являются обращением непосредственно к ОС. Все остальные библиотечные функции, которые используются в ОС (**scanf()**, **printf()**, **fscanf()**, **fprintf()**, **fgets()**, **fputs()**, **getc()**, **putc()**, **fgetc()**, **fputc()**), работают с буферизацией. Т.е. СВ **read()** и **write()** — единственные, которые не используют буферизацию.

В лекции 3.5 уже говорилось о том, что нормальное завершение программы осуществляется тремя функциями: СВ **_exit()** и **_Exit()**, сразу же возвращающими управление ядру, и библиотечной функцией **exit()**, выполняющей ряд дополнительных операций, таких, как принудительный сброс всех буферов ввода-вывода, и только после этого возвращающей управление ядру. Все три функции закрывают все открытые в данном процессе файлы.

В системе UNIX имена **_exit()** и **_Exit()** являются синонимами, оба СВ не сбрасывают буферы ввода-вывода. Т.е. использующие буферизацию функции, такие как **printf()**, ничего не смогут вывести и вернут признак ошибки, тогда как небуферизованный вывод через СВ **write()** будет выводиться всегда.

Поэтому при выводе информации через буферизированные функции нужно использовать функцию **exit()**, являющуюся оберткой для СВ **_exit()**. Можно также с помощью СВ **fflush()** принудительно сбросить все буферы «вручную» для того, чтобы выводы функций, использующих буферизацию, успели появиться на экране до выхода из программы через СВ **_exit()**:

```
fflush(NULL);
```

Итак:

- Не использующий буферизацию СВ **write()** выводит информацию всегда и в первую очередь.
- Функции вывода, использующие буферизацию, такие как **printf()**, ничего не выведут при закрытии программы посредством СВ **_exit()** и **_Exit()**, т.к. последние не сбрасывают буферы ввода-вывода.
- Функции вывода, использующие буферизацию, осуществляют успешный вывод при закрытии программы со сбросом буферов ввода-вывода посредством функции **exit()** или комбинации СВ **fflush()** и СВ **_exit()**. При этом информация, выводимая через СВ **write()**, не использующий буферизацию, будет выведена в первую очередь, т.е. перед той информацией, которая выводится через функции, использующие буферизацию.

Пример 3.1

Влияние завершения процесса через библиотечную функцию **exit()** на последовательность вывода информации через функции, использующие буферизацию (**printf()**) и через СВ **write()**, не использующий буферизацию.

lect74-3.1.c:

```
// Заголовочные файлы:
#include <stdlib.h> // Для библиотечной функции exit()
#include <stdio.h> /* Для поддержания стандартного ввода-вывода. Включает функции
                     scanf(), printf(), fgets(), fputs() и др. Функции stdio
                     наслаждаются поверх СВ ввода-вывода (open(), close(),
                     read(), write() и т.д.). */
#include <unistd.h> /* Для возможности мобильного переноса с одной вычислительной
                     платформы на другую */

void main()
{
    /* На экран (в стандартный поток вывода) посредством функции printf() через
       буферизированный вывод выводится цепочка символов: */
    printf("\nHello");
    /* СВ write() записывает в файл стандартного вывода цепочку заданной длины
       (9 байт) из символьного массива "my friend": */
}
```

```

write(1, "my friend", 9);
/* Библиотечная функция exit() сбрасывает буферы ввода-вывода и завершает
процесс, закрывая все открытые в процессе файлы: */
exit(0);
}

```

Ответ:

```

my friend
Hello

```

Пример 3.2

Влияние завершения процесса через СВ **_exit()** на отсутствие вывода информации через функции, использующие буферизацию (**printf()**) и на вывод информации через СВ **write()**, не использующий буферизацию.

lect74-3.2.c:

```

// Заголовочные файлы:
#include <stdio.h>      /* Для поддержания стандартного ввода-вывода. Включает
функции
                           scanf(), printf(), fgets(), fputs() и др. Функции stdio
                           налагаются поверх СВ ввода-вывода (open(), close(),
                           read(), write() и т.д.). */
#include <unistd.h>    /* Для возможности мобильного переноса с одной вычислительной
платформы на другую */

void main()
{
    /* На экран (в стандартный поток вывода) посредством функции printf() через
       буферизированный вывод выводится сообщение: */
    printf("\nHello");
    /* СВ write() записывает в файл стандартного вывода цепочку заданной длины
       (9 байт) из символьного массива "my friend": */
    write(1, "my friend", 9);
    /* СВ _exit() немедленно завершает процесс, закрывая все открытые в процессе
       файлы, но не выталкивая буферы ввода-вывода: */
    _exit(0);
}

```

Ответ:

```

my friend

```

8. Межпроцессный (программный) канал в ОС UNIX. Примеры реализации

Стивен Р.У., Раго С.А. UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018 г., стр. 616:

Неименованные каналы (pipes, или программные каналы, или коммуникационные каналы, далее просто межпроцессные каналы) — это старейшая форма организации взаимодействий между процессами, предоставляемая ОС UNIX. Каналы имеют два ограничения:

1. Исторически они являются **полудуплексными** (то есть данные могут передаваться по ним только в одном направлении). Некоторые современные системы предоставляют дуплексные каналы, но для сохранения переносимости приложений никогда не следует пользоваться этой возможностью.
2. Каналы могут использоваться только для организации взаимодействий между процессами, которые имеют общего предка. Обычно канал создается родительским процессом, который затем вызывает СВ `fork()`, после чего канал может использоваться для общения между родительским и дочерним процессами. Т.е. между процессами, которые знают идентификаторы друг друга, например **процесс-отец/процесс-сын, процесс-сын/процесс-внук, процесс-дед/процесс-правнук** и т.п.

Тип файла межпроцессного (программного) канала — **FIFO**. Такие файлы предназначены для создания/открытия временного буфера (канала), обеспечивающего взаимодействие двух или нескольких процессов посредством записи данных в этот буфер и чтения данных из буфера. Это односторонние (полудуплексные) файлы. FIFO файлы работают по схеме «first-in-first-out» (первый пришел — первый ушел). Буфер, связанный с FIFO файлом, создается, когда один из процессов открывает этот файл для чтения или записи. **Межпроцессный канал — байтовый** (3 входящих «сообщения» на выходе сливаются в одно большое сообщение).

При открытии файла межпроцессного канала:

- Открывается сразу два пользовательских дескриптора файла — один для чтения из канала, а другой для записи в канал.
- Файл канала создается в ОП в пространстве пользователя, поэтому с помощью него удобно осуществлять обмен информацией между параллельно работающими процессами.
- Диспозиция файла канала — рабочий, он существует до тех пор, пока от него не отделяются все процессы, которые связаны с ним. Т.е. буфер удаляется, когда все процессы, связанные с FIFO файлом межпроцессного канала, закрывают все свои ссылки на него или когда завершится процесс, создавший этот файл.

8.1 Системный вызов `pipe()`

Системный вызов **pipe()** создает коммуникационный канал межпроцессной связи между двумя взаимосвязанными процессами:

```
#include <unistd.h>
int pipe (int fd[2]);
```

Через аргумент **fd** возвращаются два пользовательских файловых дескриптора межпроцессного канала: **fd[0]** открыт для чтения из канала, а **fd[1]** — для записи в канал.

Данные, выводимые в **fd[1]**, становятся входными данными для **fd[0]**.

СВ `pipe()` возвращает **0** в случае успеха, **-1** — в случае ошибки.



Как уже было сказано выше, межпроцессные каналы могут использоваться только для организации взаимодействий между процессами, которые имеют общего предка. Обычно канал создается родительским

процессом, который затем вызывает СВ **fork()**, после чего канал может использоваться для общения между родительским и дочерним процессами. Т.е. между процессами, которые знают идентификаторы друг друга, например процесс-отец/процесс-сын, процесс-сын/процесс-внук, процесс-дед/процесс-правнук и т.п.

Основная идея, заложенная в межпроцессный канал, заключается в том, чтобы процесс создал своего потомка для того, чтобы потомок выполнил для данного процесса какую-то работу и результат этой работы передал через межпроцессный канал процессу-родителю. Обычно после этого процесс-потомок становится не нужен и закрывается, а процесс-родитель пользуется результатами работы процесса-потомка.

Создание межпроцессного канала

Общепринятый метод создания каналов состоит в следующем: программа создает канал по системному вызову **pipe()**, после чего разделяется на две копии с помощью СВ **fork()**. Затем в одном из полученных процессов закрывается сторона канала для чтения — **fd[0]**, а в другом процессе — сторона канала для записи — **fd[1]**. После чего по созданному одностороннему каналу производится обмен информацией между родительским и дочерним процессами.



lect81-1.c:

```

// Заголовочные файлы
#include <unistd.h> /* Для возможности мобильного переноса с одной вычислительной
                     платформы на другую */
void main()
{
    // Выделение памяти под переменные:
    int fd[2]; /* Для хранения пользовательских файловых дескрипторов
                  межпроцессного канала, открываемого СВ pipe(). */

    /* Для каждого процесса изначально открыты три основных файла потоков, которые
       Могут использовать программы, они идентифицируются по номеру
       пользовательского дескриптора файла:
       STDIN или 0 - этот файл связан с клавиатурой, и большинство команд получают
                      данные для работы отсюда;
       STDOUT или 1 - это стандартный вывод, сюда программа отправляет все
                      результаты своей работы. Он связан с экраном или, если быть
                      точным, с терминалом, в котором выполняется программа;
       STDERR или 2 - все сообщения об ошибках выводятся в этот файл.

    СВ pipe() создает межпроцессный канал и открывает два пользовательских
    Дескриптора файлов - fd[0] с доступом к каналу на чтение и fd[1] с доступом
    на запись, которые будут записаны в ТПДОФ (таблицу пользовательских
    дескрипторов открытых файлов) процесса под новыми номерами 3 и 4,
    соответственно: */

    pipe(fd);
    /* С помощью СВ fork() текущий процесс разделяется на две идентичные копии,
       которые продолжают выполняться как два независимых процесса:
       1) Процесс-потомок получает от СВ fork() код ответа 0.
       2) Процесс-родитель - идентификатор, под которым запущен процесс-потомок.
       В случае аварийного завершения СВ fork() вернет -1. */
    if (fork() == 0)
    { // Ветка для потомка после успешного вызова fork():
        close(fd[1]); // Закрываем запись
        write(fd[0], "Hello, world!", 13); // Пишем в канал
        close(fd[0]); // Закрываем чтение
    }
    else
    {
        close(fd[0]); // Закрываем чтение
        read(fd[1], buffer, 13); // Читаем из канала
        close(fd[1]); // Закрываем запись
    }
}
  
```

```

// Закрытие файлового дескриптора канала, предназначенного для записи:
close(fd[1]);
// Теперь потомок может только читать информацию от родителя...
}
else
{
    // Ветка для родителя после успешного вызова fork():
    // Закрытие файлового дескриптора канала, предназначенного для чтения:
    close(fd[0]);
    // Теперь родитель может только передавать информацию потомку...
}
/* Примечание: чтобы организовать передачу в обратном направлении, родительский
процесс закрывает fd[1], а дочерний процесс - fd[0]. */
}

```

8.2 Синхронизация обмена информации через межпроцессный канал

Как уже было сказано выше, **исторически межпроцессные каналы являются полудуплексными** (то есть данные могут передаваться по ним только в одном направлении). **Некоторые современные системы предоставляют дуплексные каналы, но для сохранения переносимости приложений никогда не следует пользоваться этой возможностью, лучше предпочесть иной способ межпроцессного взаимодействия.** Если все же решено использовать межпроцессный канал в дуплексном режиме, то передача должна быть строго синхронизирована: нельзя лить воду в трубу (pipe) одновременно с двух концов, иначе в какой-то момент она «лопнет».

Рассмотрим синхронизацию обмена через межпроцессный канал, работающий в полудуплексном режиме.

За синхронизацией действий записи и чтения из канала следит сама ОС.

Если процесс (в данном случае — сын) читает пустой канал, то он будет ждать появления данных, пока кто-нибудь не запишет данные в этот канал. Если в канале осталось много не считанной информации, то записывающий процесс (отец) будет ждать освобождения места в канале.



Когда один из концов канала закрывается, в силу вступают следующие два правила:

1. **Если у канала доступ на запись закрыт, то при чтении из такого канала СВ `read()` вернет код ответа 0, чтобы сообщить о достижении конца файла после того, как все данные будут прочитаны.**



Технически признак конца файла не будет сгенерирован, пока не будут закрыты все дескрипторы, открытые для записи в канал. Такое возможно при создании дубликатов дескрипторов, благодаря чему сразу несколько родственных процессов могут производить запись в канал. Однако обычно у канала имеется один дескриптор, открытый для чтения, и один дескриптор, открытый для записи.

2. **Если у канала доступ на чтение закрыт, а записывающий процесс (отец) пытается записать данные в канал, то он (отец) получит от ядра ОС сигнал SIGPIPE (разорванный канал), который завершит выполнение процесса-отца. Если приложение игнорирует этот сигнал или перехватывает его и возвращает управление из обработчика сигнала нормальным образом, СВ `write()` вернет значение -1 и код ошибки EPIPE в переменной errno .**



8.3-8.4 Передача информации через межпроцессный канал. Пример реализации

Пример 1

lect83-1.c:

```

// Заголовочные файлы
#include <stdio.h>          /* Для поддержания стандартного ввода-вывода. Включает
                               функции scanf(), printf(), fgets(), fputs() и др.
                               Функции stdio наслаждаются поверх СВ ввода-вывода
                               (open(), close(), read(), write() и т.д.). */
#include <sys/types.h>
#include <unistd.h>           /* Для возможности мобильного переноса с одной
                               вычислительной платформы на другую */
#include <fcntl.h>             // Для управления файлами
#include <sys/wait.h>          /* Для СВ wait(), позволяющего процессу-родителю дождаться
                               момента, когда процесс-потомок выполнит работу и пришлет
                               результаты своей работы */

void main()
{
    /* Выделение памяти под переменные:
       p[2] - для хранения пользовательских файловых дескрипторов межпроцессного
       канала, открываемого СВ pipe().
       s      - для хранения причины гибели процесса-сына, записанного СВ wait().
       k      - для сохранения кода возврата СВ read(). */

    int p[2], s, k;
    char buf[80];      // Символьный массив-буфер
    /* Для каждого процесса изначально открыты три основных файла потоков, которые
       Могут использовать программы, они идентифицируются по номеру
       пользовательского дескриптора файла:
       STDIN или 0 - этот файл связан с клавиатурой, и большинство команд получают
                      данные для работы отсюда;
       STDOUT или 1 - это стандартный вывод, сюда программа отправляет все
                      результаты своей работы. Он связан с экраном или, если быть
                      точным, с терминалом, в котором выполняется программа;
       STDERR или 2 - все сообщения об ошибках выводятся в этот файл.

       ТПДОФ процесса:
    
```

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| | | |

СВ pipe() создает межпроцессный канал и открывает два пользовательских дескриптора файлов - p[0] с доступом к каналу на чтение и p[1] с доступом на запись, которые будут записаны в ТПДОФ (таблицу пользовательских дескрипторов открытых файлов) процесса под новыми номерами 3 и 4, соответственно:

ТПДОФ процесса после СВ pipe() и межпроцессный канал:

| | | | |
|------|---|-----------------------|-----|
| | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[0] | 3 | pipe _r | ... |
| p[1] | 4 | pipe _w | ... |
| | 5 | | |

p[0] Процесс
p[1] отец

*/

pipe(p);

/* С помощью СВ fork() текущий процесс разделяется на две идентичные копии, которые продолжают выполняться как два независимых процессы:

- 1) Процесс-потомок получает от СВ fork() код ответа 0.
 - 2) Процесс-родитель - идентификатор, под которым запущен процесс-потомок.
- В случае аварийного завершения СВ fork() вернет -1.
- Оба процесса параллельные, идентичные, включая переменные, регистры и ТПДОФ (у родителя и потомка открыты одни и те же файлы). У них есть общий файл канала.

У каждого из процессов в ТПДОФ есть два пользовательских дескриптора с номерами p[0] и p[1], соответствующих этому файлу.

ТПДОФ процессов после СВ fork():

Отец

| | | | |
|------|---|-----------------------|-----|
| | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[0] | 3 | pipe _r | ... |
| p[1] | 4 | pipe _w | ... |
| | 5 | | |

Сын

| | | | |
|------|---|-----------------------|-----|
| | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[0] | 3 | pipe _r | ... |
| p[1] | 4 | pipe _w | ... |
| | 5 | | |

p[0] Процесс
p[1] отец

Процесс
сын p[0]
 p[1]

<limits.h>

*/

if (fork() != 0)

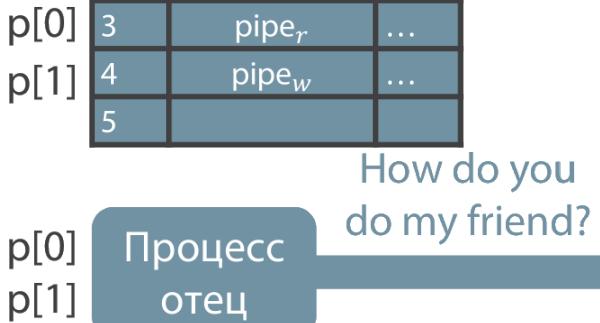
{ // Ветка для родителя после успешного вызова fork():

/* Процесс-родитель выполняет СВ wait() и ждет завершения выполнения процесса-потомка.

ТПДОФ процесса-отца после выполнения СВ wait():

Отец

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | pipe _w | ... |
| 5 | | |



```
*/
wait(&s); // s = 512
/* Дождавшись, когда процесс-потомок завершится, процесс-родитель может
прочитать
информацию из канала и проанализировать ее.
СВ wait() возвращает идентификатор завершившегося процесса-потомка
(в данном примере идентификатор не запоминается, т.к. порождается всего
один потомок и нет нужды различать, какой именно потомок завершил свое
существование) и в свой единственный аргумент s записывает причину гибели
процесса-потомка.

В ОС семейства UNIX процесс может завершиться либо с помощью сигнала
(сигнал - программная версия аппаратных прерываний), либо по СВ _exit().
Т.к. процесс-потомок завершается с помощью СВ _exit(), который завершится
с аргументом 2, следовательно s = 512, т.к. аргумент СВ _exit()
записывается с 8 по 15 биты (биты с 0 по 7 заполняются тогда, когда
процесс убивается сигналом и в них записываются номер сигнала и действия
процесса по сигналу). Следовательно, двойка, записанная в двоичной
системе исчисления в 9 и 8 биты:
s = 00000010 0 0000000 (2) = 512 (10)
```

Значение переменной s = 512₁₀

| Аргумент exit() | | | | | | | | | | | | | | | | Номер сигнала | | | | | | | |
|-----------------|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---------------|----|----|----|----|----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |

```
*/
/* Закрытие файлового дескриптора канала, предназначенного для записи.
ТПДОФ процесса-отца после закрытия канала на запись:
```

Отец

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | | |
| 5 | | |

p[0]

How do you
do my friend?

p[0]

Процесс
отец

*/

close(p[1]);

// Теперь родитель может только читать информацию от потомка...

/* СВ read() читает из межпроцессного канала цепочку заданной длины (80 байтов), состоящую из любых символов, и записывает их в буфер buf. Переменная k будет хранить число байт, успешно прочитанных СВ read() из межпроцессного канала:

p[0]

Процесс
отец

How do you
do my friend?

*/

k = read(p[0], buf, 80);

/*

Межпроцессный канал после того, как была прочитана вся информация, содержащаяся в нем:

p[0]

Процесс
отец

*/

/* СВ write() записывает в файл стандартного вывода цепочку заданной длины (k байт) из символьного массива-буфера buf: */

write(1, buf, k); // buf = How do you do my friend?

/* На экран (в стандартный поток вывода) посредством функции printf() через буферизированный вывод выводится количество байт, прочитанных из межпроцессного канала: */

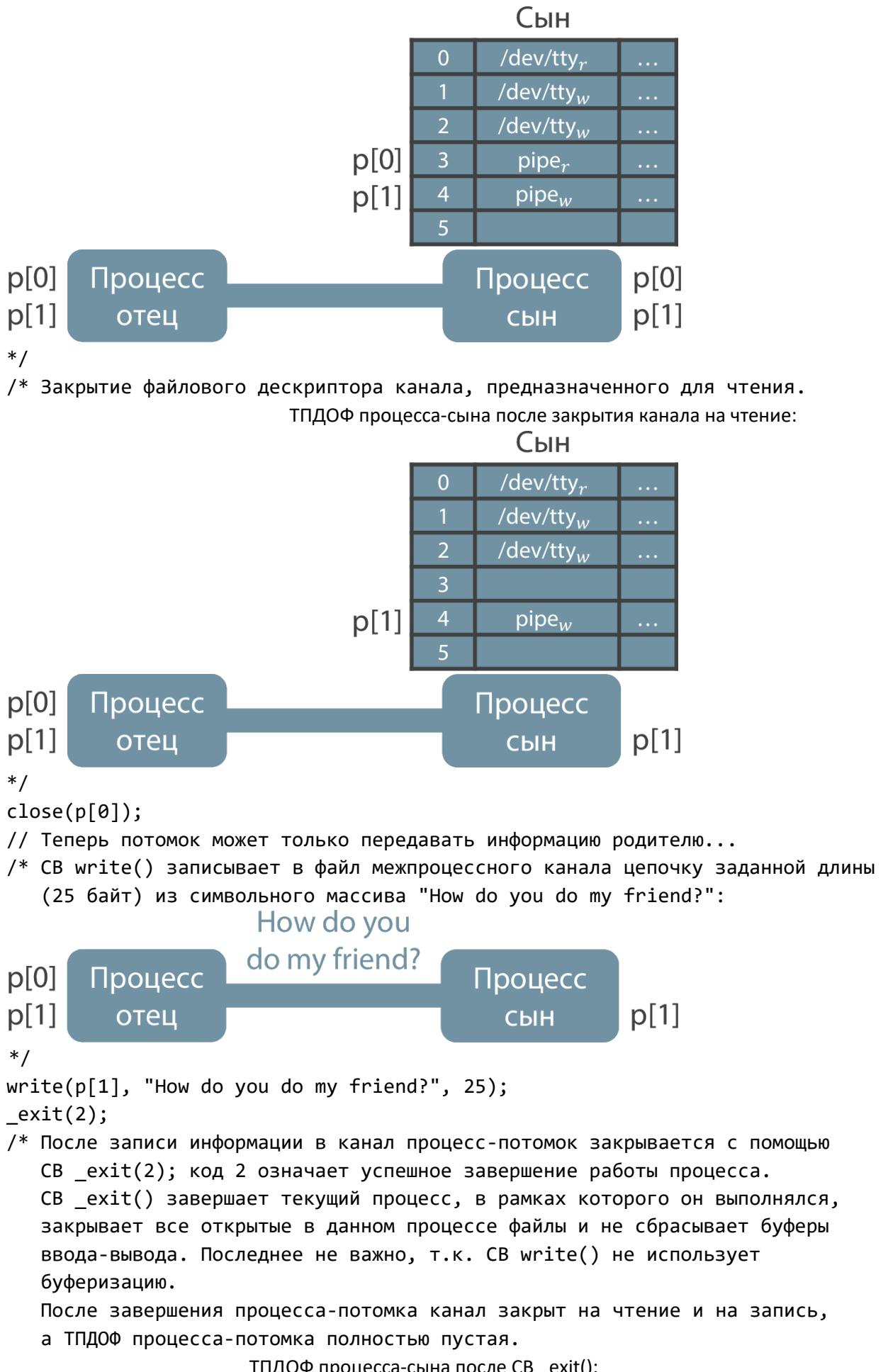
printf("k = %d\n", k); // k = 25

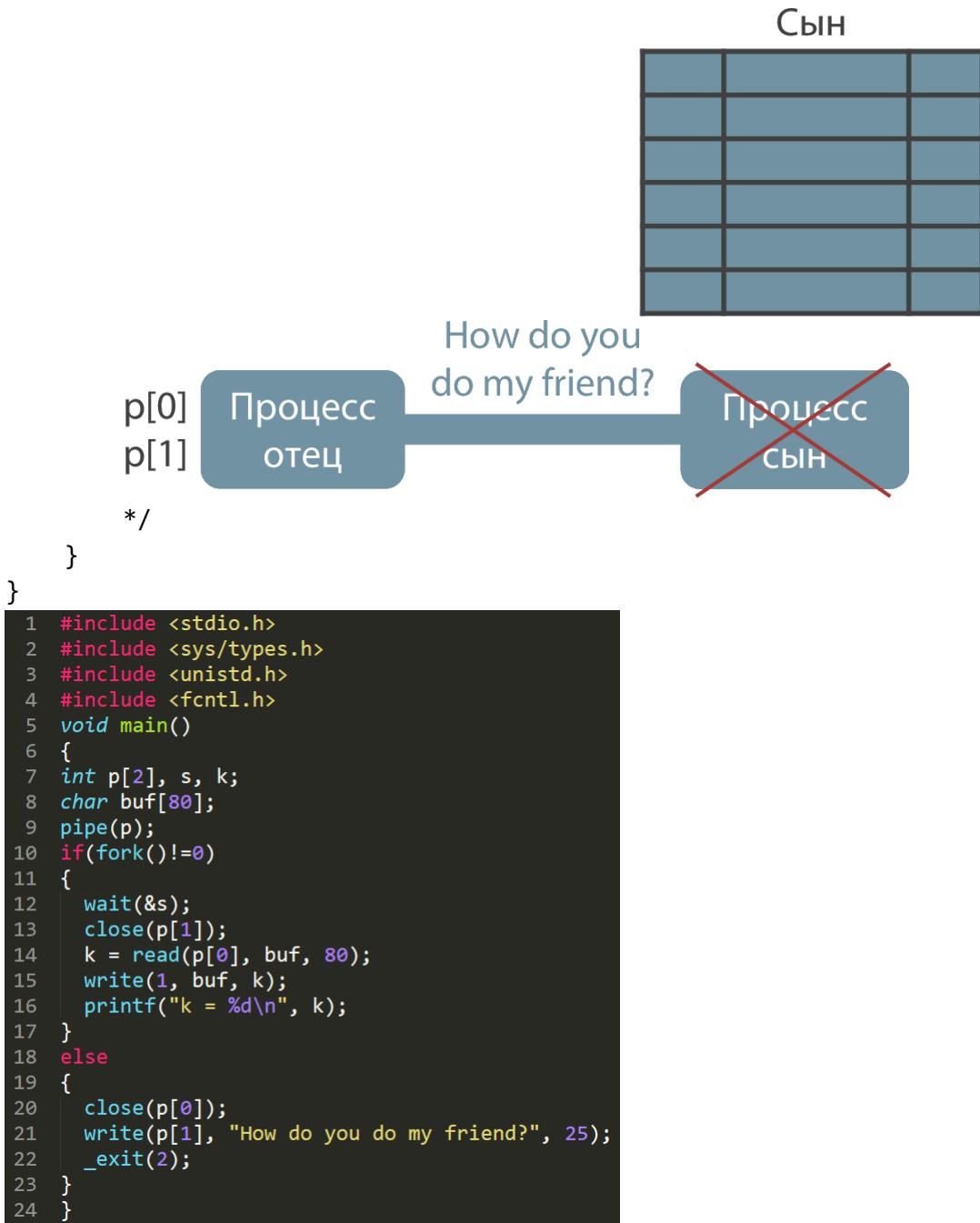
/* При завершении процесса все открытые им файлы автоматически закрываются ядром. Многие приложения используют это обстоятельство и не закрывают файлы явно. Закрытие файла приводит также к снятию любых блокировок, которые могли быть наложены процессом. */

}

else

{ /* Ветка для потомка после успешного вызова fork():
ТПДОФ процесса-сына:





Ответ:

How do you do my friend?k = 25

Пример 2

lect83-2.c

// Заголовочные файлы

```
#include <stdio.h>
```

```
/* Для поддержания стандартного ввода-вывода. Включает  
функции scanf(), printf(), fgets(), fputs() и др.  
Функции stdio наслаждаются поверх СВ ввода-вывода  
(open(), close(), read(), write() и т.д.). */
```

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
/* Для возможности мобильного переноса с одной  
вычислительной платформы на другую */
```

```
#include <sys/wait.h>
```

```
/* Для СВ wait(), позволяющего процессу-родителю дождаться момента, когда процесс-потомок выполнит работу и пришлет результаты своей работы */
```

```

#include <string.h>      /* Содержит прототипы функций работы со строками string(),
                           strcasecmp(), strcat(), strcpy(), memset(), bzero() и
т.д. */
#include <stdlib.h>       // Для библиотечной функции exit()

void main()
{
    /* Выделение памяти под переменные:
       fd[2] - для хранения пользовательских файловых дескрипторов межпроцессного
       канала, открываемого СВ pipe().
       stat - для хранения причины гибели процесса-сына, записанного СВ wait().
       n - для сохранения кода возврата СВ read(). */
    int fd[2], stat, n;
    char buf[80]; // Символьный массив-буфер
    /* Поскольку могут быть предоставлены грязные страницы памяти, то прежде чем
       использовать выделенную память, желательно ее очистить с помощью функции
       memset() или bzero(), занося нулевые значения в область памяти: */
    memset(buf, 80, 0);
    /* Для каждого процесса изначально открыты три основных файла потоков, которые
       Могут использовать программы, они идентифицируются по номеру
       пользовательского дескриптора файла:
       STDIN или 0 - этот файл связан с клавиатурой, и большинство команд получают
                      данные для работы отсюда;
       STDOUT или 1 - это стандартный вывод, сюда программа отправляет все
                      результаты своей работы. Он связан с экраном или, если быть
                      точным, с терминалом, в котором выполняется программа;
       STDERR или 2 - все сообщения об ошибках выводятся в этот файл.

       ТПДОФ процесса:
    
```

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |

СВ pipe() создает межпроцессный канал и открывает два пользовательских дескриптора файлов - fd[0] с доступом к каналу на чтение и fd[1] с доступом на запись, которые будут записаны в ТПДОФ (таблицу пользовательских дескрипторов открытых файлов) процесса под новыми номерами 3 и 4, соответственно:

ТПДОФ процесса после СВ pipe():

| | | |
|-------|-----------------------|-------------------|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| fd[0] | 3 | pipe _r |
| fd[1] | 4 | pipe _w |
| 5 | | |

```

*/
pipe(fd);
/* С помощью СВ fork() текущий процесс разделяется на две идентичные копии,

```

которые продолжают выполняться как два независимых процесса:

- 1) Процесс-потомок получает от СВ fork() код ответа 0.
 - 2) Процесс-родитель - идентификатор, под которым запущен процесс-потомок.
- В случае аварийного завершения СВ fork() вернет -1.
- Оба процесса параллельные, идентичные, включая переменные, регистры и ТПДОФ (у родителя и потомка открыты одни и те же файлы). У них есть общий файл канала.

У каждого из процессов в ТПДОФ есть два пользовательских дескриптора с номерами fd[0] и fd[1], соответствующих этому файлу.

ТПДОФ процессов после СВ fork():

| Отец | | | Сын | | |
|-------|-----------------------|-------------------|-------|-----------------------|-------------------|
| 0 | /dev/tty _r | ... | 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... | 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... | 2 | /dev/tty _w | ... |
| fd[0] | 3 | pipe _r | fd[0] | 3 | pipe _r |
| fd[1] | 4 | pipe _w | fd[1] | 4 | pipe _w |
| | 5 | | | 5 | |

*/

```
if (fork() == 0)
{
    // Ветка для потомка после успешного вызова fork():
    /* Закрытие файлового дескриптора канала, предназначенного для чтения:
       ТПДОФ процесса-сына после закрытия канала на чтение:
```

Сын

| | | |
|-------|-----------------------|-------------------|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | | |
| fd[1] | 4 | pipe _w |
| | 5 | |

*/

```
close(fd[0]);
```

// Теперь потомок может только передавать информацию родителю...

```
/* Закрытие дескриптора файла стандартного вывода STDOUT_FILENO = 1:
```

ТПДОФ процесса-сына после закрытия файла стандартного вывода:

Сын

| | | |
|-------|-----------------------|-------------------|
| 0 | /dev/tty _r | ... |
| 1 | | |
| 2 | /dev/tty _w | ... |
| 3 | | |
| fd[1] | 4 | pipe _w |
| | 5 | |

*/

```
close(1);
```

/* Отождествление стандартного вывода с файловым дескриптором канала, предназначенным для записи. Для этого посредством СВ dup() выполняется

копирование дескриптора канала, предназначенного для записи, в первую (наименьшую) свободную запись в ТПДОФ (таблице пользовательских дескрипторов открытых файлов) процесса (начиная от 0), т.е. в STDOUT_FILENO = 1, поскольку STDIN_FILENO = 0 занят, а дескриптор файла стандартного вывода с номером 1 был предварительно закрыт.

```
*/
dup(fd[1]);
/* Закрытие лишней копии файлового дескриптора канала, предназначенного
   для записи: */
close(fd[1]);
/* Осуществлено перенаправление вывода. Вместо файла стандартного вывода,
   которым является монитор, установлен межпроцессный канал, который открыт
   на запись, поэтому результат выполнения любой стандартной инструкции,
   функции или утилиты будет выводиться не на экран монитора, а в
   межпроцессный канал.
```

ТПДОФ процесса-сына после выполнения перенаправления вывода:

Сын

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | pipe _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | | |
| 4 | | |
| 5 | | |

В межпроцессный канал (в стандартный поток вывода вместо экрана) посредством функции printf() через буферизированный вывод выводится сообщение: */

```
printf("Получите информацию\n");
exit(1);
/* Библиотечная функция exit() сбрасывает буфера ввода-вывода и завершает
   процесс, закрывая все открытые в процессе файлы; код 1 означает успешное
   завершение работы процесса. Благодаря сбрасыванию буферов, буфер от
   функции printf() будет вытолкнут в канал. После завершения процесса-
   потомка канал закрыт на чтение и на запись и ТПДОФ процесса-потомка
   полностью пустая.
```

ТПДОФ процесса-сына после функции exit():

Сын

| | | |
|--|--|--|
| | | |
| | | |
| | | |
| | | |
| | | |

Получите
информацию\n

fd[0] Процесс
 отец

Процесс
 ~~сын~~

```

}

else
{ // Ветка для родителя после успешного вызова fork():
/* Данная часть кода процесса-родителя выполняется параллельно с выполнением
процесса-потомка:
Закрытие файлового дескриптора канала, предназначенного для записи.
Теперь родитель может только читать информацию от потомка...
ТПДОФ процесса-отца после закрытия канала на запись:

    Отец


|       |                       |     |
|-------|-----------------------|-----|
| 0     | /dev/tty <sub>r</sub> | ... |
| 1     | /dev/tty <sub>w</sub> | ... |
| 2     | /dev/tty <sub>w</sub> | ... |
| fd[0] | pipe <sub>r</sub>     | ... |
| 4     |                       |     |
| 5     |                       |     |


*/
close(fd[1]);
/* Процесс-родитель выполняет СВ wait() и ждет завершения выполнения
процесса-потомка. */
wait(&stat); // stat = 256
/* Дождавшись, когда процесс-потомок завершится, процесс-родитель может
прочитать информацию из канала и проанализировать ее.
СВ wait() возвращает идентификатор завершившегося процесса-потомка
(в данном примере идентификатор не запоминается, т.к. порождается всего
один потомок и нет нужды различать, какой именно потомок завершил свое
существование) и в свой единственный аргумент stat записывает причину
гибели процесса-потомка.

В ОС семейства UNIX процесс может завершиться либо с помощью сигнала
(сигнал - программная версия аппаратных прерываний), либо по СВ _exit().
Т.к. процесс-потомок завершается с помощью библиотечной функции exit(),
которая завершается с аргументом 1, следовательно stat = 256, т.к.
аргумент функции exit() записывается с 8 по 15 биты (биты с 0 по 7
заполняются тогда, когда процесс убивается сигналом и в них записываются
номер сигнала и действия процесса по сигналу). Следовательно, единица,
записанная в двоичной системе исчисления в 8 бит:
stat = 00000001 0 0000000 (2) = 256 (10)

```

Значение переменной stat = 256₁₀

Межпроцессный канал после выполнения СВ wait():



/* СВ read() читает из межпроцессного канала цепочку заданной длины (80 байтов), состоящую из любых символов, и записывает их в буфер buf. Переменная n будет хранить число байт, успешно прочитанных СВ read() из межпроцессного канала.

Поскольку в файле межпроцессного канала меньше, чем 80 байт, то СВ read() прочитает все данные и вернет число меньше 80, а указатель чтения/записи уткнется в конец файла:

Межпроцессный канал после прочтения из него всей информации:

fd[0] Процесс отец

```
/*
n = read(fd[0], buf, 80);
/* На экран (в стандартный поток вывода) посредством функции printf() через
буферизированный вывод выводится количество байт, прочитанных из
межпроцессного канала: */
printf("n = %d\n", n); /* n = 20, но под Ubuntu Linux n = 38 (зависит
от кодировки) */

/* После того, как уже были прочитаны все данные, при повторной попытке
чтения встретится признак конца файла и СВ read() вернет 0. Указатель
чтения/записи все так же будет установлен в конец файла: */
n = read(3, &buf[n + 1], 80);
/* На экран (в стандартный поток вывода) посредством функции printf() через
буферизированный вывод выводятся: количество байт, прочитанных из
межпроцессного канала во второй раз; причина гибели процесса-потомка. */
printf("n = %d stat = %d\n", n, stat); // n = 0 stat = 256

/* После закрытия пользовательского дескриптора канала, предназначенного для
чтения, межпроцессный канал будет удален, т.к. все процессы, связанные
с FIFO файлом межпроцессного канала, закрыли все свои ссылки на него:
ТПДОФ процесса-отца после закрытия (удаления) канала:
```

Отец

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | | |
| 4 | | |
| 5 | | |

Процесс отец

```
/*
close(fd[0]);
/* При завершении процесса все открытые им файлы автоматически закрываются
ядром. Многие приложения используют это обстоятельство и не закрывают
файлы явно. Закрытие файла приводит также к снятию любых блокировок,
которые могли быть наложены процессом. */

}
```

```

}
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <stdlib.h>
4 void main()
5 {
6 int fd[ 2 ], stat, n;
7 char buf[ 80 ];
8 memset( buf, 80, 0 );
9 pipe( fd );
10 if( fork() == 0 )
11 {
12     close( fd[ 0 ] );
13     close( 1 );
14     dup( fd[ 1 ] );
15     close( fd[ 1 ] );
16     printf( "Получите информацию\n" );
17     exit( 1 );
18 }
19 else
20 {
21     close( fd[ 1 ] );
22     wait( &stat );
23     n = read( fd[ 0 ], buf, 80 );
24     printf( "n = %d\n", n );
25     n = read( 3, buf, 80 );
26     printf( "n = %d stat = %d\n", n, stat );
27     close( fd[ 0 ] );
28 }
29 }
```

Ответ:

n = 20, но под Ubuntu Linux n = 38 (зависит от кодировки)

n = 0 stat = 256

9. Обработка прерываний в ОС UNIX

UNIX-подобные ОС относятся к классу **ОС с разделением времени**, т.е. процессорное время делится между несколькими параллельно работающими процессами. В таких системах реакция процессов на любое предусмотренное в вычислительной системе аппаратное прерывание была бы неоднозначной, поэтому в таких системах реализована программная версия аппаратных прерываний, которые называются **сигналами**.

9.1 Понятие сигнала. Типы сигналов в ОС UNIX

Сигнал — это программное средство, с помощью которого может быть прервано функционирование процесса в UNIX-подобной ОС.

Механизм сигналов позволяет процессам реагировать на различные события, которые могут произойти в ходе функционирования процесса внутри него самого, или во внешней среде. Причем события могут быть различного типа, даже не предусмотренные в рамках ОС.

События, инициирующие сигнал

1. **Введение пользователем управляющих символов с терминала** (например, SIGINT (код 2) — сигнал прерывания <Ctrl+C> с терминала, вызывающий завершение процесса; SIGTSTP (код 20) — сигнал остановки с терминала <Ctrl+Z>, приводящий к остановке процесса);
2. **Возникновение аварийной ситуации при функционировании процесса** (например, SIGSEGV (код 11) обращение к памяти, находящейся за пределами адресного пространства процесса, приводящий к завершению процесса с дампом памяти; SIGFPE (код 8) деление на ноль или другая неправильная операция с плавающей точкой, приводящий к завершению процесса с дампом памяти);
3. **Возникновение заранее описанного события** (например, в программе написано, что через 5 секунд процессу нужно отправить сигнал побудки);
4. **Возникновение не предусмотренного или не поддающегося идентификации события** (например, отказ какого-то аппаратного блока — сбой по питанию, отключение питания и т.д.).

Сигналы поддерживаемые ОС

Информация о сигналах, поддерживаемые ОС UNIX, отображена в заголовочном файле <signal.h>, а также Стивен Р.У., Раго С.А. UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018 г., стр. 380-389.

Пояснения к таблице:

- Название — символьная константа, обозначающая сигнал.
- Код — номер сигнала.
- Действие по умолчанию — как ОС обработает получение процессом того или иного сигнала.
- Описание — причина посылки сигнала.

У каждой UNIX-подобной ОС перечень сигналов может чуть-чуть отличаться, но в целом они повторяются по номерам и по символьным константам, поэтому переход между ОС не будет сопровождаться какими-то сложностями.

Например, из таблицы видно, что причиной посылки сигнала SIGXCPU (код 30) является зацикливание процесса. SIGXCPU посыпается, когда время, потраченное процессором для выполнения процесса, превышает допустимое значение. Процессорное время, это не время, прошедшее с запуска процесса, а только то время, в течение которого процессор занимался его выполнением (в остальное время процесс ожидает ввода-вывода и обслуживание системой других процессов).

Также, чтобы предотвратить либо случайный, либо умышленный монопольный захват ресурсов компьютера каким-то одним процессом, ОС устанавливает в специальном аппаратном таймере прерываний временной интервал (квант времени), в течение которого любому процессу разрешается занимать ЦП.

Если процесс добровольно не освобождает ЦП в течение указанного временного интервала, таймер вырабатывает сигнал прерывания, по которому управление будет передано ОС. После этого ОС переведет ранее выполнявшийся процесс в состояние готовности, а первый процесс из списка готовых — в состояние выполнения.

Выжимка из файла <signal.h>:

| Название | Код | Действие по умолчанию | Описание |
|-----------|-----|----------------------------|----------------------------------------------------------|
| SIGABRT | 6 | Завершение с дампом памяти | Сигнал посылаемый функцией <code>abort()</code> |
| SIGBUS | 10 | Завершение с дампом памяти | Неправильное обращение в физическую память |
| SIGCHLD | 18 | Игнорируется | Дочерний процесс завершен или остановлен |
| SIGCONT | 25 | Продолжить выполнение | Продолжить выполнение ранее остановленного процесса |
| SIGFPE | 8 | Завершение с дампом памяти | Ошибкачная арифметическая операция |
| SIGHUP | 1 | Завершение | Закрытие терминала |
| SIGILL | 4 | Завершение с дампом памяти | Недопустимая инструкция процессора |
| SIGINT | 2 | Завершение | Сигнал прерывания (Ctrl-C) с терминала |
| SIGKILL | 9 | Завершение | Безусловное завершение |
| SIGPIPE | 13 | Завершение | Запись в разорванное соединение (пайл, сокет) |
| SIGQUIT | 3 | Завершение с дампом памяти | Сигнал «Quit» с терминала (Ctrl-\) |
| SIGSEGV | 11 | Завершение с дампом памяти | Нарушение при обращении в память |
| SIGSTOP | 23 | Остановка процесса | Остановка выполнения процесса |
| SIGTERM | 15 | Завершение | Сигнал завершения (сигнал по умолчанию для утилиты kill) |
| SIGTSTP | 20 | Остановка процесса | Сигнал остановки с терминала (Ctrl-Z). |
| SIGTTIN | 26 | Остановка процесса | Попытка чтения с терминала фоновым процессом |
| SIGTTOU | 27 | Остановка процесса | Попытка записи на терминал фоновым процессом |
| SIGUSR1 | 16 | Завершение | Пользовательский сигнал № 1 |
| SIGUSR2 | 17 | Завершение | Пользовательский сигнал № 2 |
| SIGPOLL | 22 | Завершение | Событие, отслеживаемое <code>poll()</code> |
| SIGPROF | 29 | Завершение | Истечение таймера профилирования |
| SIGSYS | 12 | Завершение с дампом памяти | Неправильный системный вызов |
| SIGTRAP | 5 | Завершение с дампом памяти | Ловушка трассировки или брейкпоинт |
| SIGURG | 21 | Игнорируется | На сокете получены срочные данные |
| SIGVTALRM | 28 | Завершение | Истечение «виртуального таймера» |
| SIGXCPU | 30 | Завершение с дампом памяти | Процесс превысил лимит процессорного времени |
| SIGXFSZ | 31 | Завершение с дампом памяти | Процесс превысил допустимый размер файла |

Реакция процесса на сигнал

1. **Игнорирование сигнала.** Получив сигнал процесс может его просто проигнорировать.
2. **Выполнение стандартной (установленной в ОС) программы обработки сигнала.** Получив сигнал процесс может вызвать на выполнение некоторую программу, аналогичную программе обработки прерывания (т.е. передать обработку этого сигнала ОС).
3. **Выполнение программы обработки, написанной пользователем.** Получив сигнал процесс может выполнить свою обработку данного сигнала.

Выполнив программу обработки, процесс продолжает свое функционирование с той точки исходной программы, в которой получил соответствующий сигнал.

+ Понятие прерывания. Схемы прерываний.

Вход в ядро ОС осуществляется по **прерыванию**. Когда ядро реагирует на данное прерывание, оно запрещает все другие прерывания.

После определения причины данного прерывания, ядро передает его обработку специальному системному процессу, предназначенному для работы с прерываниями этого типа.

Схема прерываний отличается для различных конфигураций компьютера. В целом, можно выделить 6 основных типов прерываний:

1. По вызову супервизора.

Прерывание по вызову супервизора появляется в том случае, когда работающий процесс выполняет команду обращения к супервизору. Этой командой (SVC-командой) программа пользователя генерирует запрос на предоставление конкретной системной услуги, например, на выполнение операции ввода-вывода, увеличение объема выделенной памяти и т.п.

2. Прерывание ввода-вывода.

Инициируется аппаратурой ввода-вывода. Например, когда завершается операция ввода-вывода, возникает ошибка или происходит смена состояния устройства ввода-вывода.

3. Внешние прерывания.

Причиной могут служить различные события, в том числе истечение кванта времени, заданного на таймере, нажатие клавиши прерывания на клавиатуре (<Ctrl+C> и др.), прием сигнала прерывания от другого процессора в мультипроцессорной системе.

4. Прерывание по рестарту.

Происходит, когда нажата кнопка рестарта (reset), или когда от другого процессора в мультипроцессорной системе поступает команда рестарта.

5. Прерывание по ошибке программы.

Например, при делении на 0.

6. Прерывание по ошибке компьютера.

Например, вышел из строя блок питания.

Для обработки каждого типа прерываний в составе ОС предусмотрены программы, называемые обработчиками прерываний.

Когда происходит прерывание, ОС запоминает состояние прерванного процесса и передает управление соответствующему обработчику прерываний. Когда обработчик прерываний завершается, ЦП начинает обслуживать либо тот процесс, который выполнялся в момент прерывания, либо процесс из списка готовых с наивысшим приоритетом, в зависимости от того, допускает ли процесс перехват ЦП.

9.2 Поддержка сигналов ядром ОС UNIX

СВ работы с сигналами

Системные вызовы `signal()` и `sigaction()` предоставляют процессу возможность самостоятельно определить свою реакцию на получение сигнала. СВ `signal()` — это простейший интерфейс к сигналам UNIX. Эта функция обеспечивает обратную совместимость с приложениями, требующими устаревшей семантики. **Новые приложения не должны использовать недостоверные сигналы.**

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

где:

`sig` — символьная константа или номер сигнала (целое число) из таблицы, представленной выше;

`*func()` — указатель на функцию обработки сигнала, которая не возвращает значение и принимает

единственный целочисленный аргумент. В качестве аргумента `*func()` можно передать:

- Константу `SIG_DFL` или `0`, которая связывает с сигналом действие по умолчанию (третья колонка в таблице выше).
- Константу `SIG_IGN` или `1`, которая сообщает системе, что сигнал должен игнорироваться. (Учтите, что два сигнала `SIGKILL` и `SIGSTOP` не могут игнорироваться и перехватываться, об этом написано ниже).

- Адрес функции, которая будет вызываться при получении сигнала, то есть будет «перехватывать» сигнал. Такие функции называются **обработчиками или перехватчиками сигналов**. Функция-обработчику сигнала передается единственный аргумент (целое число — номер сигнала), и она ничего не возвращает. Эта функция должна быть описана или объявлена перед вызовом СВ **signal()**.

Когда СВ **signal()** вызывается, чтобы установить обработчик сигнала, второй аргумент должен быть указателем на функцию. Возвращает значение, полученное в результате последнего вызова **func()** (т.е. предыдущую диспозицию сигнала или результат предыдущей обработки сигнала) в случае успеха, в случае ошибки возвращает **-1**.

СВ **sigaction()** позволяет проверить действие, связанное с определенным сигналом, изменить его или выполнить обе эти операции. Этот СВ с расширенным функционалом (поддержкой сигнальной маски процесса) служит заменой СВ **signal()** из ранних версий UNIX. Поскольку семантика СВ **signal()** различается в разных реализациях, вместо него следует использовать СВ **sigaction()**.

Сигнал SIGKILL прекращает выполнение процесса. Это довольно специфический сигнал, который посыпается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнал процесса. Иногда он также посыпается ОС (например, при завершении работы системы). **Сигналы SIGSTOP и SIGKILL предназначены только для остановки или завершения процесса и не могут игнорироваться или перехватываться, то есть обрабатываться при помощи определенной пользователем процедуры.**

Пример использования СВ **signal()**

```
signal(SIGHUP, function);
```

Где :

- SIGHUP — это символьное имя сигнала (код 1), в данном случае он посыпается управляющему процессу (лидеру сеанса), связанному с управляющим терминалом , если обнаружен обрыв связи с терминалом.
- function может принимать следующие значения:
 1. 0 (SIG_DFL) — сигнал обрабатывает ОС;
 2. 1 (SIG_IGN) — игнорировать сигнал;
 3. Имя функции (точнее, ее адрес) написанной пользователем, причем эта функция должна быть описана или объявлена перед вызовом СВ **signal()**.

Обработка сигналов в ОС UNIX

Пример 1

Установить для сигнала SIGINT (код 2) обработчик по умолчанию (сигнал обрабатывает ОС), вызывающий завершение процесса при нажатии < Ctrl+C >:

```
#include <signal.h>
signal( SIGINT, 0 );      // signal( 2, SIG_DFL );
```

Пример 2

Игнорировать сигнал SIGINT (код 2). После этого завершить процесс нажатием < Ctrl+C > будет невозможно.

```
#include <signal.h>
signal( SIGINT, 1 );      // signal( SIGINT, SIG_IGN );
```

Пример 3

Установить для сигнала SIGINT (код 2) функцию **bizon()** — пользовательский обработчик сигнала. Функция должна быть описана или объявлена перед вызовом СВ **signal()**.

```
#include <signal.h>
void bizon();              // объявление функции обработчика
signal( SIGINT, bizon );  // signal( 2, bizon );
```

Программная реализация

lect92-1.c:

```
// Заголовочные файлы:
#include <signal.h> // Для использования своего обработчика сигналов
#include <stdlib.h> // Для CB system()
void main()
{
    /* Функция обработки сигнала handl() должна быть описана или объявлена
       перед использованием CB signal() или sigaction(): */
    void aaa(); // Объявление функции обработчика сигнала
    /* CB signal() должен быть объявлен в начальных инструкциях программы,
       иначе в начале программы ОС будет применять обработчик по умолчанию.
       Установка функции aaa() в качестве пользовательского обработчика
       сигнала SIGINT (код 2): */
    signal(SIGINT, aaa);
    /* Тело программы.
       Каждое нажатие комбинации клавиш < Ctrl+C > во время выполнения
       программы сгенерирует сигнал прерывания SIGINT, после чего будет
       выполнена программа обработчика, затем процесс продолжит свое
       функционирование с той точки исходной программы, в которой получил
       соответствующий сигнал.
       Бесконечный цикл для того, чтобы пользователь успел нажать < Ctrl+C >:
    */
    while (1);
}
void aaa() // Описание функции обработчика сигнала
{
    /* CB system() позволяет вызывающей программе выполнить произвольную
       консольную команду или вызвать другую программу, не выходя из
       контекста вызывающего процесса. При нажатии на < Ctrl+C > CB system()
       запускает утилиту date, которая выводит на монитор текущую дату
       и время, т.е. дату и время поступления сигнала. */
    system("date");
}
```

9.3 Сигнальная маска процесса в ОС UNIX

CB **sigaction()** обладает расширенным функционалом (поддержкой сигнальной маски процесса) и служит заменой CB **signal()** из ранних версий UNIX. CB **sigaction()** позволяет проверить действие, связанное с определенным сигналом, изменить его или выполнить обе эти операции. Для того, чтобы изучить, как работает CB **sigaction()**, нужно понять, что представляет собой сигнальная маска процесса в ОС UNIX.

Сигнальная маска процесса

Сигнальная маска есть у каждого процесса — это один из атрибутов (одно из полей) дескриптора процесса в ОС UNIX. Т.е. каждая запись таблицы процессов (дескриптор процесса) ядра ОС содержит сигнальную маску соответствующего процесса.

Сигнальная маска — массив сигнальных флагов, отражающая, какие сигналы процесс ожидает. Маска указывает, какие сигналы (сигнал — программная версия аппаратных прерываний и все аппаратные прерывания реализованы в виде сигналов) игнорируются, а какие перехватываются.

По своей сути это битовая переменная (тип этой переменной описан в заголовочном файле **signal.h**), которая содержит количество бит, равное количеству сигналов, поддерживаемых ОС. Так, если ОС

поддерживает 40 сигналов, то размерность сигнальной маски равна 40 битам. Каждый бит этой маски соответствует номеру того сигнала, который представлен в ОС. Например:

- 1 бит — характеризует сигнал SIGHUP (код 1);
- 2 бит — характеризует сигнал SIGINT (код 2);
- и т.д.

Например, если значение бита в сигнальной маске (например, 2-ого бита) равно 1, то процесс ожидает сигнал с соответствующим кодом (SIGINT) и обязан прореагировать на его поступление. Если же значение бита в сигнальной маске равно 0, то процесс игнорирует этот сигнал.

UNIX-подобные ОС относятся к классу ОС с разделением времени, т.е. процессорное время делится между несколькими параллельно работающими процессами. В таких системах реакция процессов на любое предусмотренное в вычислительной системе аппаратное прерывание была бы неоднозначной, поэтому в таких системах реализована программная версия аппаратных прерываний, которые называются сигналами.

Сигнальная маска процесса — это особенность, которая характеризует ОС с разделением времени.

Как отреагировать на сигнал процессу, который в данный момент находится в очереди на выполнение? За счет сигнальной маски производится такое перераспределение, описываемое обычно в литературе как «**блокирование сигнала**», но на самом деле это — **отложенная обработка сигнала**.

Пусть значение бита в сигнальной маске процесса равно единице. Это означает, что процесс ожидает **данный сигнал**. Однако в момент получения сигнала процесс может не работать, а ждать, когда ОС предоставит ему процессор для выполнения. Т.е. процесс отработал свой квант времени, и, находясь в очереди, ожидает, когда ОС предоставит ему очередной квант времени, а сигнал приходит в это время. Поскольку в сигнальной маске процесса стоит единица, то процесс ожидает этот сигнал и он обязан будет прореагировать на этот сигнал тогда, когда получит процессор в свое распоряжение. Если же значение бита равно нулю, то процесс просто не заметит этого сигнала и проигнорирует его.

Почему **системы с разделением времени нельзя использовать для процессов, описываемых в реальном времени?** Потому что обработка сигналов может быть отложенная, тогда как в системах реального времени на сигнал требуется реагировать моментально.

Наследование сигнальной маски процесса

При создании процесс наследует сигнальную маску своего родителя, но не один сигнал, ожидающий обработки родителем, к процессу-потомку не пропускается.

Маска сигналов наследуется у родительского процесса при СВ `fork()` и `exec()`. Поскольку при создании процесс наследует сигнальную маску своего родителя, то весь перечень сигналов, которые поддерживает ОС, каким-то образом отражен в сигнальной маске процесса и ОС на него адекватно должна реагировать.

9.4 Системные вызовы манипулирования сигнальной маской процесса. Часть 1

Перед тем, как изучить работу СВ `sigaction()`, нужно научиться работать с сигнальной маской процесса.

Процесс может получить текущее значение сигнальной маски (СМ), изменить существующую СМ (либо задать новую) или выполнить сразу обе операции с помощью СВ:

```
#include <signal.h> // Описан тип битовой переменной sigset_t, которая характеризует СМ процесса
```

```
int sigprocmask(int cmd, const sigset_t*new, sigset_t*old);
```

cmd определяет операцию над СМ процесса:

- **SIG_SETMASK** — заменяет СМ значением, указанным в **new**. Т.е. новая маска сигналов в аргументе **new** замещает текущую маску сигналов. Третий аргумент не нужен (старая маска не нужна) и может быть пустым указателем. Например, так маска текущего процесса заменяется значением из переменной **mask**: `sigprocmask(SIG_SETMASK, &mask, NULL)`

- **SIG_BLOCK** — добавляет сигналы (устанавливает в единицу биты), указанные в **new**, в СМ. Это означает, что аргумент **new** содержит дополнительные сигналы, которые требуется заблокировать.
- **SIG_UNBLOCK** — удаляет из СМ (делает нулевыми) сигналы, указанные в **new**. Это означает, что аргумент **new** содержит сигналы, которые требуется разблокировать.
- **0** — не изменять СМ процесса. Также, если в аргументе **new** передается пустой указатель, маска сигналов процесса не изменяется и значение аргумента **cmd** игнорируется. Нужно для получения СМ текущего процесса. Например, так в переменную **mask** записывается СМ текущего процесса: `sigprocmask(0, NULL, &mask)`.

new определяет набор сигналов, которые будут либо сделаны новой СМ процесса (**SIG_SETMAK**), либо будут добавлены в (**SIG_BLOCK**), либо удалены из (**SIG_UNBLOCK**) СМ вызывающего процесса.

old — это адрес переменной, которой присваивается СМ вызывающего процесса до вызова `sigprocmask()`. Т.е. в эту переменную сохраняется прежняя маска процесса (например, для последующего восстановления).

Возвращает **0** в случае успеха, и **-1** в случае неудачи.

Обратите внимание:

- **Сигналы SIGKILL и SIGSTOP нельзя заблокировать, игнорировать или перехватить.**
- Функция `sigprocmask` определена только для однопоточных процессов. Для работы с масками сигналов в многопоточных приложениях предоставляются отдельные функции.

СВ, позволяющие управлять битами в СМ процесса

Обратите внимание, что объявив переменную **mask** типа `sigset_t` и занеся в нее нужные данные с помощью одной или нескольких указанных ниже функций, пока вы не примените СВ `sigprocmask()`, сигнальная маска процесса не установится! Т.е. поменять СМ процесса можно только с помощью СВ `sigprocmask()`, а все указанные ниже функции служат для того, чтобы сформировать набор битов, которые вы будете либо добавлять, либо удалять, либо заменять СМ процесса на этот набор битов.

```
#include <signal.h> // Описан тип битовой переменной sigset_t, которая характеризует СМ процесса
```

```
int sigemptyset(sigset_t*mask);
```

Сбрасывает (обнуляет) все биты в переменной **mask**.

Возвращает **0** в случае успеха, и **-1** — в случае неудачи.

Обычно после выделения памяти под переменную **mask** нужно обнулить эту память, т.к. могут быть выделены грязные страницы памяти, для этого и используется функция `sigemptyset()`.

```
#include <signal.h>
```

```
int sigfillset(sigset_t*mask);
```

Устанавливает в 1-цу все биты сигналов в маске **mask**. Антипод функции `sigemptyset()`. Использование этой функции позволяет процессу реагировать на все сигналы, которые поддерживает данная ОС.

Возвращает **0** в случае успеха, и **-1** — в случае неудачи.

```
#include <signal.h>
```

```
int sigaddset(sigset_t*mask, int sig);
```

Добавляет (устанавливает в единицу) бит, соответствующий сигналу **sig** в маске **mask**. В качестве аргумента **sig** можно использовать либо номер (код) сигнала, либо соответствующую ему символьную константу.

Возвращает **0** в случае успеха, и **-1** — в случае неудачи.

```
#include <signal.h>
```

```
int sigdelset(sigset_t*mask, int sig);
```

Сбрасывает (обнуляет) бит, соответствующий сигналу **sig** в маске **mask**. Т.е. в дальнейшем процесс будет игнорировать заданный сигнал. Антипод функции **sigaddset()**.

Возвращает **0** в случае успеха, и **-1** — в случае неудачи.

```
#include <signal.h>
int sigismember(sigset_t*mask, int sig);
```

Проверяет состояние указанного разряда (установлен ли в маске **mask** единичный бит для сигнала **sig**). Т.е. с помощью этой функции можно узнать, ожидает ли процесс заданного сигнала, или не ожидает. Функция удобна для того, чтобы проверить, как процесс будет реагировать на тот или иной сигнал. Поскольку сигнал с номером **0** отсутствует, при определении номера разряда из номера сигнала вычитается **1**.

Возвращает **1** (истина, процесс ожидает этот сигнал), **0** (ложь) и **-1** — в случае ошибки.

9.5 Системные вызовы манипулирования сигнальной маской процесса. Часть 2

Пример

Проверить наличие флага сигнала SIGINT в сигнальной маске процесса, и, если флаг нулевой, то установить его и удалить сигнал SIGSYS из сигнальной маски процесса.

lect95-1.c:

```
// Заголовочные файлы:
#include <stdio.h> // Для поддержания стандартного ввода-вывода
#include <signal.h> /* Для работы с сигнальной маской (СМ) процесса -
                     описан тип битовой переменной sigset_t */
#include <stdlib.h> // Для библиотечной функции exit()

int main()
{
    // Выделение памяти под битовую переменную mask (40-45 бит, 5-6 байт):
    sigset_t mask;
    // Очищение этой памяти на случай, если выделены "грязные" страницы:
    sigemptyset(&mask);
    // В переменную mask записывается СМ текущего процесса:
    if (sigprocmask(0, 0, &mask) == -1)
    { /* Если произошла ошибка, завершается выполнение процесса.
        В стандартный поток вывода (на экран) посредством функции printf()
        через буферизированный вывод выводится сообщение об ошибке: */
        printf("Ошибка\n");
    /* Библиотечная функция exit() сбрасывает буфера ввода-вывода
       и завершает процесс, закрывая все открытые в процессе файлы;
       код 1 означает завершение работы завершение работы процесса,
       вызванное ошибкой СВ sigprocmask(). Благодаря сбрасыванию буферов,
       буфер от функции printf() будет вытолкнут в стандартный файл
       вывода. */
        exit(1);
    }
    else
    { /* Если функция sigprocmask() завершилась успешно, то в СМ процесса
        проверяется бит для сигнала SIGINT (код 2): */
        if (sigismember(&mask, SIGINT) == 0)
            /* Если этот бит нулевой, то с помощью функции sigaddset() этот
               бит устанавливается в единичку: */

```

```

        sigaddset(&mask, SIGINT);
    }
// Удаление (обнуление) бита для сигнала SIGSYS (код 12):
sigdelset(&mask, SIGSYS);
/* После выполненных операций над битами переменной mask, текущая
СМ процесса устанавливается в соответствии с этими битами.
Аргумент SIG_SETMASK - новая маска сигналов mask замещает
текущую маску сигналов, третий аргумент не нужен (старая маска
не нужна) и равен нулю (пустому указателю): */
if (sigprocmask(SIG_SETMASK, &mask, 0) == -1)
    // Если произошла ошибка, завершается выполнение процесса:
    printf("Ошибка 2\n");
return 0; // Код 0 означает успешное завершение работы процесса.
}

```

```

1 #include <stdio.h>
2 #include <signal.h>
3 #include <stdlib.h>
4 int main()
5 {sigset_t mask;
6  sigemptyset( &mask);
7  if(sigprocmask( 0, 0, &mask ) == -1)
8      { printf( "Ошибка\n" ); exit( 1 ); }
9  else
10     { if( sigismember( &mask, SIGINT ) == 0 ) sigaddset( &mask, SIGINT ); }
11     sigdelset( &mask, SIGSYS );
12     if( sigprocmask( SIG_SETMASK, &mask, 0 ) == -1 ) printf( "Ошибка 2\n" );
13     return 0;
14 }

```

10. Системные вызовы и программы (утилиты) обработки прерываний в ОС UNIX. Примеры реализаций

10.1 Системный вызов `sigaction()`

СВ `sigaction()` позволяет проверить действие, связанное с определенным сигналом, изменить его или выполнить обе эти операции. Этот СВ с расширенным функционалом (поддержкой сигнальной маски процесса) служит заменой СВ `signal()` из ранних версий UNIX. Поскольку семантика СВ `signal()` различается в разных реализациях, вместо него следует использовать СВ `sigaction()`. Способы работы с сигнальной маской процесса посредством СВ `sigprocmask()` и нескольких других функций мы уже изучили.

СВ `sigaction()` позволяет процессу установить свою реакцию на получение сигнала.

```
#include <signal.h>          /* Описание типа данных struct sigaction */
int sigaction(int sig, struct sigaction *new,
              struct sigaction *old);
```

где:

- Через аргумент `sig` передается номер или символьная константа сигнала, который будет обработан в рамках данного процесса.
- Аргумент `new` — новый метод обработки указанного сигнала. Если в аргументе `new` передается непустой указатель, диспозиция сигнала изменяется.
- Аргумент `old` — старый метод обработки указанного сигнала. Если в аргументе `old` передается непустой указатель, функция возвращает предыдущее значение диспозиции сигнала, помещая его по указанному в `old` адресу.

Возвращает 0 в случае успеха, -1 — в случае ошибки

Этот СВ использует следующую структуру:

```
struct sigaction
{
    void (*sa_handler)(int); /* имя (адрес) функции-обработчика сигнала,
                               указанного в 1-м аргументе sig,
                               или SIG_IGN = 1 (игнорировать сигнал),
                               или SIG_DFL = 0 (обработка сигнала по умолчанию) */
    sigset_t sa_mask;        /* Задает СМ текущего процесса. При этом бит сигнала,
                               соответствующего аргументу sig, устанавливается в СМ
                               равным 1, т.е. процесс будет ожидать данный сигнал
                               */
    int sa_flag;             /* флаги - действия по обработке сигнала, которые
                               предусматривает ОС наряду с обработчиком данного
                               сигнала */
};
```

Поле `sa_flags` показывает, какое действие по обработке сигнала, указанного в аргументе `sig`, определено в аргументе `new`:

- 0 — Стандартная обработка остальных сигналов. Т.е. обработка будет соответствовать СВ `signal()`. Единственное отличие от СВ `signal()` состоит в том, что вызвать СВ `sigaction()` нужно только один раз и затем все дальнейшие реакции на сигнал будут обрабатываться с помощью метода, указанного в `sigaction()`. Т.е. после установки диспозиции сигнала она остается неизменной, пока явно не изменится вызовом СВ `sigaction()`.

- **SA_NOCLDSTOP** — Если **sig** равен **SIGCHLD**, то уведомление об остановке дочернего процесса не будет получено (в тех случаях, когда дочерний процесс получает сигнал **SIGSTOP**, **SIGTSTP**, **SIGTTIN** или **SIGTTOU**). Этот флаг используется редко.
- **SA_ONESHOT** или **SA_RESETHAND** — Восстановить реакцию на сигнал по умолчанию после одного вызова обработчика.
- **SA_RESTART** — Производить автоматический перезапуск системных вызовов, прерванных данным сигналом. Реакция на сигналы должна соответствовать семантике сигналов ОС BSD (также Mac OS, Solaris и др.) и позволять некоторым системным вызовам работать, в то время как идет обработка сигналов. Флаг полезен при переносе программ с одной вычислительной системы на другую.
- **SA_NOMASK** или **SA_NODEFER** — Не препятствовать получению сигнала при его обработке. Т.е. не блокировать сигнал автоматически при вызове функции-обработчика (если, конечно, сигнал не включен в маску **sa_mask**). Заметьте, что **такое поведение соответствует поведению ненадежных сигналов в ранних версиях UNIX**.

В поле **sa_flags** можно побитно складывать несколько флагов, если это специально оговорено в задании. Если нет особых требований по обработке сигнала, то лучше использовать стандартный метод обработки, т.е. в поле **sa_flags** использовать значение **0**.

Стивен Р.У., Раго С.А. *UNIX. Профессиональное программирование*. 3-е издание. — СПб.: Питер, 2018 г., стр. 418:

*Если при изменении диспозиции сигнала поле **sa_handler** содержит адрес функции-обработчика (а не константы **SIG_IGN** или **SIG_DEL**), то поле **sa_mask** определяет набор сигналов, которые будут добавлены к маске сигналов процесса перед вызовом функции-обработчика. Перед возвратом из обработчика сигнала маска сигналов будет автоматически восстановлена в прежнее состояние. Таким способом можно блокировать определенные сигналы на время работы функции-обработчика. Перед доставкой сигнала, когда вызывается функция-обработчик, сам сигнал также включается в маску сигналов; тем самым блокируется доставка того же самого сигнала на время выполнения обработчика. Заблокированные сигналы обычно не помещаются в очередь. Если сигнал был сгенерирован пять раз за период времени, когда он был заблокирован, функция-обработчик, как правило, вызывается только один раз.*

*После установки диспозиции сигнала она остается неизменной, пока явно не изменится вызовом СВ **sigaction()**. В отличие от ранних версий UNIX с их ненадежными сигналами, стандарт POSIX.1 требует, чтобы действие сигнала оставалось неизменным, пока явно не будет изменено программой.*

10.2 Пример реализации обработки сигналов на процедурном языке программирования

Пример. Обработка сигнала SIGINT.

lect102-1.c:

```
// Заголовочные файлы:
#include <stdio.h> // Для поддержания стандартного ввода-вывода
#include <signal.h> /* Для работы с сигнальной маской процесса и макросами,
                     прототипами данных из функции sigaction() */

/* Глобальная переменная n, видимая всем функциям данной программы,
   выполняет роль счетчика сигналов: */
int n = 0;

/* Функция обработки сигнала func() должна быть описана или
   объявлена перед использованием СВ signal() или sigaction(): */
void func()
```

```

{
    /* Тело программы обработчика */
    /* Счетчик сигналов прерывания увеличивается при поступлении
       каждого сигнала, и на экран выводится сообщение: */
    n = n + 1;
    printf("Поступил сигнал №: %d\n", n);
}

int main()
{
    /* Настройка обработчика сигнала: */

    /* Выделяется память под два объекта типа struct sigaction, каждый объект
       такого типа содержит 3 поля. В объекте n будет указан новый метод
       обработки сигнала, а в объекте o - сохранен старый метод обработки
       сигнала, который в коде данной программы не используется. */
    struct sigaction n, o;
    /* Могут быть предоставлены грязные страницы памяти, поэтому используя
       функцию sigemptyset сбрасываются (обнуляются) все биты в переменной sa_mask
       из объекта n. В эту переменную затем будет записана сигнальная маска (СМ)
       текущего процесса. Если СМ не записывать, а оставить очищенную переменную
       sa_mask как есть, то дополнительны обрабатываемых сигналов к СМ процесса
       перед вызовом функции-обработчика func() добавлено не будет.

       Напомним, что если определенный бит маски установлен, соответствующий
       ему сигнал будет обработан. Например, если значение бита в СМ (например
       2-ого бита) равно 1, то процесс ожидает сигнал с соответствующим кодом
       (SIGINT) и обязан прореагировать на его поступление.

       Если же значение бита в СМ равно 0, то процесс игнорирует этот сигнал. */
    sigemptyset(&n.sa_mask);
    /* Задаются поля структуры struct sigaction:
       Сохранение текущей СМ. Первые 2 аргумента нулевые, т.е. СМ текущего
       процесса не будет изменяться, а будет записана в переменную n.sa_mask. */
    sigprocmask(0, 0, &n.sa_mask);
    /* Поскольку при создании процесс-потомок наследует СМ своего родителя, то
       весь перечень сигналов, которые поддерживаются ОС, отражен в СМ процесса,
       и ОС должна на него адекватно реагировать. */
    /* В поле sa_handler указывается имя (адрес) функции обработки сигнала,
       описанной в начале программы: */
    n.sa_handler = func;
    /* Поле sa_flags = 0, т.е. предусматривается стандартная обработка сигнала,
       который будет указан в СВ sigaction(). */
    n.sa_flags = 0;
    /* После того, как заполнены все три поля нового метода обработки сигнала n,
       исполняется СВ sigaction(), первым аргументом которого является SIGINT
       (код 2) - сигнал прерывания < Ctrl+C > с терминала, по умолчанию
       вызывающий завершение процесса.

       Вторым аргументом является n - новый метод обработки сигнала.

       Третий аргумент o означает, что старый метод обработки сигнала будет
       сохранен в объекте o, хотя он и не будет использоваться в дальнейшем.

       Поле n.sa_mask определяет набор сигналов, которые будут добавлены к СМ
   */
}

```

процесса перед вызовом функции-обработчика `n.sa_handler = func()`. Перед возвратом из обработчика сигнала СМ будет автоматически восстановлена в прежнее состояние. Таким способом можно блокировать определенные сигналы на время работы функции-обработчика.

Перед доставкой сигнала, когда вызывается функция-обработчик, сам сигнал также включается в СМ; тем самым на время выполнения обработчика блокируется доставка того же самого сигнала. Такой подход гарантирует то, что во время обработки последующее поступление определенных сигналов будет приостановлено до завершения вызова. */

```

if (sigaction(SIGINT, &n, &o) == -1)
    printf("Ошибка в sigaction\n");
/* СВ sigaction() описан в начале программы, поэтому, если в дальнейшем
поступит сигнал SIGINT, то программа всегда на него отреагирует: любое
нажатие < Ctrl+C > на клавиатуре вызовет функцию-обработчик func(), которая
при каждом прерывании будет увеличивать счетчик сигналов, которые получил
данный процесс и выводить значение счетчика на экран. */

/* Тело основной программы: */
/* Бесконечный цикл для того, чтобы пользователь успел нажать < Ctrl+C >: */
while (1);
return 0; // Код 0 означает успешное завершение работы процесса.
}

```

```

1 int main()
2 { struct sigaction n, o;
3     sigemptyset( &n.sa_mask );
4     sigprocmask( 0, 0, &n.sa_mask );
5     n.sa_handler = func;
6     n.sa_flag = 0;
7     if( sigaction( SIGINT, &n, &o ) == -1 )
8         printf( "Ошибка в sigaction\n" );
9     /* тело программы
10     .... */
11     return 0;
12 }

```

10.3 Библиотечные функции безусловного нелокального перехода

Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX: Руководство программиста по разработке ПО — М.: ДМК Пресс, 2000 г., стр. 158-159:

Иногда при получении сигнала необходимо вернуться на предыдущую позицию в программе. Например, может потребоваться, чтобы пользователь мог вернуться в основное меню программы при нажатии клавиши прерывания. Это можно выполнить, используя две специальные функции `sigsetjmp()` и `siglongjmp()`. (Существуют альтернативные функции `setjmp()` и `longjmp()`, но их нельзя использовать совместно с обработкой сигналов.) Процедура `sigsetjmp()` «сохраняет» текущие значения счетчика команд, позиции стека, регистров процессора и СМ, а процедура `siglongjmp()` передает управление назад в сохраненное таким образом положение. В этом смысле процедура `siglongjmp()` аналогична оператору `goto`. Важно понимать, что возврат из процедуры `siglongjmp()` не происходит, так как стек возвращается в сохраненное состояние. Как будет показано ниже, при этом происходит выход из соответствующего вызова `sigsetjmp()`.

Кроме навигации по меню, библиотечные функции `sigsetjmp()` и `siglongjmp()` удобно применять для случаев, когда на экране появляется запрос ввода от пользователя и следующая за ним функция чтения ввода с клавиатуры. Если перед запросом ввода от пользователя поставить функцию `sigsetjmp()`, то при получении сигнала во время ввода (пользователь ввел часть информации, но вдруг поступил сигнал), программа будет

возвращаться не на саму функцию ввода, а на этап раньше, т.е. сначала будет повторен запрос, а уже затем выполнится функция ввода. В противном случае, ситуацию с частичным вводом информации и дальнейшим поступлением сигнала, после которого программа возвращалась бы в функцию ввода, придется обрабатывать дополнительно (продолжать ввод или начинать его заново).

Библиотечная функция `sigsetjmp()`

Функция `sigsetjmp()` запоминает текущее состояние процесса, «сохраняет» текущие значения счетчика команд, позиции стека, регистров процессора и СМ. Грубо говоря, эта функция **ставит метку**.

```
#include <setjmp.h> /* Нелокальные переходы. Содержит прототипы функций
                     безусловного нелокального перехода sigsetjmp()
                     и siglongjmp(), а также определения связанных
                     с ними структур данных */
int sigsetjmp(sigjmp_buf stack, int savemask);
stack — сохраняет контекст (текущие значения счетчика команд, позиции стека, регистров процессора) процесса, а также (возможно) текущую СМ (маску блокированных сигналов и действия, связанные со всеми сигналами). Это позволяет после прерывания вернуться к последнему запомненному состоянию. Тип sigjmp_buf определен в стандартном заголовочном файле <setjmp.h>.
savemask — указывает, сохранять ли в переменную stack СМ текущего процесса (после выхода из процесса он будет продолжен со старой СМ или с новой СМ):
```

- если **savemask ≠ 0** (равно **TRUE**), то сохранять — используется чаще всего: СМ восстанавливается из сохраненного значения.
- если **savemask = 0** (равно **FALSE**), то не сохранять.

Возвращает **0**, если вызвана непосредственно (при первом вызове, т.е. при обычном порядке исполнения программы в отсутствии сигналов), и **ненулевое значение** (значение аргумента **val ≠ 0**), если возврат произошел в результате обращения к функции `siglongjmp()`.

Библиотечная функция `siglongjmp()`

Функция `siglongjmp()` передает управление назад в положение, сохраненное функцией `sigsetjmp()`. Обычно эта функция **используется в обработчике сигналов, чтобы вернуться на последнюю запомненную метку**.

```
#include <setjmp.h> /* Нелокальные переходы. Содержит прототипы функций
                     безусловного нелокального перехода sigsetjmp()
                     и siglongjmp(), а также определения связанных
                     с ними структур данных */
void siglongjmp(sigjmp_buf stack, int val);
stack — восстанавливает контекст (значения счетчика команд, позиции стека, регистров процессора и, возможно, СМ) процесса. При вызове siglongjmp(), если аргумент stack был сохранен в результате вызова sigsetjmp() с ненулевым значением savemask, СМ восстанавливается из сохраненного значения
val — то значение, которое будет возвращать последняя выполненная функция sigsetjmp(). Нужен для того, чтобы вернуться на последнее запомненное состояние в случае, когда в программе содержится несколько функций sigsetjmp(). Именно последнее запомненное состояние вернет значение, отличное от нуля, а все остальные будут возвращать ноль. Т.е. последнее запомненное состояние — это та метка, на которую нужно перейти. Поскольку это безусловный и нелокальный переход, то из любой подпрограммы будет возврат к последнему запомненному состоянию процесса, т.е. к последнему выполненному sigsetjmp().
```

10.4 Примеры реализации безусловного перехода

Пример реализации безусловного перехода, в котором используется несколько функций `sigsetjmp()` и в обработчике сигнала SIGINT используется функция `siglongjmp()`.

lect104-1.c:

```
// Заголовочные файлы:
```

```

#include <stdio.h> // Для поддержания стандартного ввода-вывода
#include <signal.h> /* Для работы с сигнальной маской процесса и макросами,
                     прототипами данных из функции sigaction() */
#include <setjmp.h> /* Нелокальные переходы. Содержит прототипы функций
                     безусловного нелокального перехода sigsetjmp()
                     и siglongjmp(), а также определения связанных
                     с ними структур данных */
#include <stdlib.h> // Для библиотечной функции system()
#include <unistd.h> /* Для возможности мобильного переноса с одной вычислительной
                     платформы на другую. В частности, для поддержания
                     функции sleep() */

/* Выделение памяти под глобальную переменную: видимая всем функциям данной
   программы переменная a1 будет сохранять состояние процесса. */
sigjmp_buf a1;

void main()
{
    /* Настройка обработчика сигнала: */
    /* Выделение памяти под объект типа struct sigaction, который содержит
       3 поля. В объекте ss будет указан новый метод обработки сигнала: */
    struct sigaction ss;
    /* Функция обработки сигнала aaa() должна быть описана или объявлена
       перед использованием СВ signal() или sigaction(): */
    void aaa(); // Объявление функции обработчика сигнала
    /* Задаются поля структуры struct sigaction:
       В поле sa_handler указывается имя (адрес) функции обработки сигнала,
       объявленной в начале программы */
    ss.sa_handler = aaa;
    /* Сохранение текущей СМ. Первые 2 аргумента нулевые, т.е. СМ текущего
       процесса не будет изменяться, а будет записана в переменную ss.sa_mask.
       Поскольку при создании процесс-потомок наследует СМ своего родителя, то
       весь перечень сигналов, которые поддерживаются ОС, отражен в СМ процесса,
       и ОС должна на него адекватно реагировать. */
    sigprocmask(0, 0, &ss.sa_mask);
    /* Поле sa_flags = 0, т.е. предусматривается стандартная обработка сигнала,
       который будет указан в СВ sigaction(). */
    ss.sa_flags = 0;
    /* После того, как заполнены все три поля нового метода обработки сигнала ss,
       исполняется СВ sigaction(), первым аргументом которого является SIGINT
       (код 2) - сигнал прерывания < Ctrl+C > с терминала, по умолчанию
       вызывающий завершение процесса.

       Вторым аргументом является ss - новый метод обработки сигнала.

       Третий аргумент 0 означает, что старый метод обработки сигнала не будет
       сохранен, т.е. он не будет использоваться в дальнейшем.

       Поле ss.sa_mask определяет набор сигналов, которые будут добавлены к СМ
       процесса перед вызовом функции-обработчика ss.sa_handler = aaa(). Перед
       возвратом из обработчика сигнала СМ будет автоматически восстановлена в
       прежнее состояние. Таким способом можно блокировать определенные сигналы
       на время работы функции-обработчика. */
}

```

Перед доставкой сигнала, когда вызывается функция-обработчик, сам сигнал также включается в СМ; тем самым на время выполнения обработчика блокируется доставка того же самого сигнала. Такой подход гарантирует то, что во время обработки последующее поступление определенных сигналов будет приостановлено до завершения вызова. */

```
sigaction(SIGINT, &ss, 0);
/* СВ sigaction() описан в начале программы, поэтому, если в дальнейшем
поступит сигнал SIGINT, то программа всегда на него отреагирует: любое
нажатие < Ctrl+C > на клавиатуре вызовет функцию обработки сигнала aaa(),
которая будет выводить текущую дату и время (т.е. время поступления
сигнала) в стандартном формате. */

/* Тело основной программы, в котором находится множество функций sigsetjmp().
Каждая функция sigsetjmp() запоминает текущее состояние (контекст)
процесса, чтобы после обработки прерывания посредством siglongjmp()
вернуться к последнему запомненному состоянию (к последнему выполненному
sigsetjmp-у).
```

При выходе из обработчика прерываний процесс будет начинаться не с того места, где поступил сигнал, а с последнего запомненного состояния процесса, т.е. с последней выполненной функции sigsetjmp().

Именно последняя запомненная функция sigsetjmp() будет всегда возвращать единичку, которая задана во втором аргументе у функции siglongjmp(), вызываемой по окончании работы обработчика. Если же функция sigsetjmp() была вызвана непосредственно (при первом вызове, т.е. при обычном порядке исполнения программы в отсутствии сигналов), то она вернет нулевое значение.

Первый аргумент a1 функции sigsetjmp() - область памяти, куда будет сохранено текущее состояние процесса, или его контекст (текущие значения счетчика команд, позиции стека, регистров процессора).

Второй аргумент 2, не равный нулю (т.е. равный TRUE) означает, что в переменную a1 будет сохраняться также и СМ (маска блокированных сигналов и действия, связанные со всеми сигналами) текущего процесса.

Это позволяет после прерывания вернуться к последнему запомненному состоянию. */

```
int ret = sigsetjmp(a1, 2); // Запоминание текущего состояния процесса
/* Тело программы...
for (int i = 0; i < 5; i++) // По сигналу цикл будет перезапущен заново
{
    printf("1st instance of sigsetjmp. Returned value = %d\n", ret);
    sleep(1); // Ждать 1 секунду, чтобы пользователь успел нажать < Ctrl+C >
}
ret = sigsetjmp(a1, 1);      // Запоминание текущего состояния процесса
/* Тело программы...
for (int i = 0; i < 5; i++) // По сигналу цикл будет перезапущен заново
{
    printf("2nd instance of sigsetjmp. Returned value = %d\n", ret);
    sleep(1); // Ждать 1 секунду, чтобы пользователь успел нажать < Ctrl+C >
}
printf("Bye!\n");
```

```

void aaa() // Описание функции обработчика сигнала
{
    /* СВ system() позволяет вызывающей программе выполнить произвольную
       консольную команду или вызвать другую программу, не выходя из
       контекста вызывающего процесса. При нажатии на < Ctrl+C > СВ system()
       запускает утилиту date, которая выводит на монитор текущую дату
       и время, т.е. дату и время поступления сигнала. */
    system("date");
    /* Безусловный нелокальный переход в последнее состояние процесса,
       сохраненное функцией sigsetjmp(). Единица - устанавливаемое возвращаемое
       значение для последней выполненной функции sigsetjmp(). */
    siglongjmp(a1, 1); // Возврат в последнее запомненное состояние процесса.
}

```

10.5 Примеры реализации безусловного перехода. Пример 2

Написать программу для ввода двух имен файлов с учетом возможного прерывания во время ввода. Реализовать безусловный переход, в котором используется несколько функций **sigsetjmp()** и в обработчике сигнала SIGINT используется функция **siglongjmp()**.

lect105-1.c:

```

// Заголовочные файлы:
#include <stdio.h> // Для поддержания стандартного ввода-вывода
#include <signal.h> /* Для работы с сигнальной маской процесса и макросами,
                     прототипами данных из функции sigaction() */
#include <setjmp.h> /* Нелокальные переходы. Содержит прототипы функций
                     безусловного нелокального перехода sigsetjmp()
                     и siglongjmp(), а также определения связанных
                     с ними структур данных */
#include <unistd.h> /* Для возможности мобильного переноса с одной вычислительной
                     платформы на другую */
#include <string.h> /* Содержит прототипы функций работы со строками string(),
                     strcasecmp(), strcat(), strcpy(), memset(), bzero() и т.д. */

/* Выделение памяти под глобальную переменную: видимая всем функциям данной
   программы переменная aa будет сохранять состояние процесса. */
sigjmp_buf aa;

/* Функция обработки сигнала pr() должна быть описана или объявлена
   перед использованием СВ signal() или sigaction(): */
void pr() // Описание функции обработчика сигнала
{
    /* Тело программы обработчика */
    printf("Поступил сигнал\n");
    /* Безусловный нелокальный переход в последнее состояние процесса,
       сохраненное функцией sigsetjmp(). Единица - устанавливаемое возвращаемое
       значение для последней выполненной функции sigsetjmp(). */
    siglongjmp(aa, 1); // Возврат в последнее запомненное состояние процесса.
}

```

```

void main()
{
    // Выделение памяти под переменные:
    /* Выделение памяти под символьные массивы для хранения двух имен файлов.
       Максимальная длина имени файла в UNIX-подобных ОС - 256 байт, включая
       завершающий нулевой символ, который показывает конец имени. */
    char name1[256], name2[256];
    /* Поскольку могут быть предоставлены грязные страницы памяти, то прежде чем
       использовать выделенную память, желательно ее очистить с помощью функции
       memset() или bzero(), занося нулевые значения в область памяти: */
    memset(name1, 0, sizeof(name1)/sizeof(name1[0]));
    memset(name2, 0, sizeof(name2)/sizeof(name2[0]));
    // bzero(name1, sizeof(name1));
    // bzero(name2, sizeof(name2));

    /* Настройка обработчика сигнала: */
    /* Выделение памяти под объект типа struct sigaction, который содержит
       3 поля. В объекте bb будет указан новый метод обработки сигнала: */
    struct sigaction bb;
    /* Задаются поля структуры struct sigaction:
       В поле sa_handler указывается имя (адрес) функции обработки сигнала,
       описанной в начале программы */
    bb.sa_handler = pr;
    /* Сохранение текущей СМ. Первые 2 аргумента нулевые, т.е. СМ текущего
       процесса не будет изменяться, а будет записана в переменную bb.sa_mask.
       Поскольку при создании процесс-потомок наследует СМ своего родителя, то
       весь перечень сигналов, которые поддерживаются ОС, отражен в СМ процесса,
       и ОС должна на него адекватно реагировать. */
    sigprocmask(0, 0, &bb.sa_mask);
    /* Поле sa_flags = 0, т.е. предусматривается стандартная обработка сигнала,
       который будет указан в СВ sigaction(). */
    bb.sa_flags = 0;
    /* После того, как заполнены все три поля нового метода обработки сигнала bb,
       исполняется СВ sigaction(), первым аргументом которого является SIGINT
       (код 2) - сигнал прерывания < Ctrl+C > с терминала, по умолчанию
       вызывающий завершение процесса.
       Вторым аргументом является bb - новый метод обработки сигнала.
       Третий аргумент 0 означает, что старый метод обработки сигнала не будет
       сохранен, т.е. он не будет использоваться в дальнейшем.
       Поле bb.sa_mask определяет набор сигналов, которые будут добавлены к СМ
       процесса перед вызовом функции-обработчика bb.sa_handler = pr(). Перед
       возвратом из обработчика сигнала СМ будет автоматически восстановлена в
       прежнее состояние. Таким способом можно блокировать определенные сигналы
       на время работы функции-обработчика.
       Перед доставкой сигнала, когда вызывается функция-обработчик, сам сигнал
       также включается в СМ; тем самым на время выполнения обработчика
       блокируется доставка того же самого сигнала. Такой подход гарантирует
       то, что во время обработки последующее поступление определенных сигналов
       будет приостановлено до завершения вызова. */
    sigaction(SIGINT, &bb, 0);
}

```

```

/* СВ sigaction() описан в начале программы, поэтому, если в дальнейшем
поступит сигнал SIGINT, то программа всегда на него отреагирует: любое
нажатие < Ctrl+C > на клавиатуре вызовет функцию обработки сигнала pr(),
которая будет выводить на экран сообщение о поступлении сигнала. */

/* Тело основной программы, в котором находится множество функций sigsetjmp().
Каждая функция sigsetjmp() запоминает текущее состояние (контекст)
процесса, чтобы после обработки прерывания посредством siglongjmp()
вернуться к последнему запомненному состоянию (к последнему выполненному
sigsetjmp-у).

При выходе из обработчика прерываний процесс будет начинаться не с того
места, где поступил сигнал, а с последнего запомненного состояния процесса,
т.е. с последней выполненной функции sigsetjmp().

Именно последняя запомненная функция sigsetjmp() будет всегда возвращать
единичку, которая задана во втором аргументе у функции siglongjmp(),
вызываемой по окончании работы обработчика. Если же функция sigsetjmp()
была вызвана непосредственно (при первом вызове, т.е. при обычном порядке
исполнения программы в отсутствии сигналов), то она вернет нулевое
значение.

Первый аргумент aa функции sigsetjmp() - область памяти, куда будет
сохранено текущее состояние процесса, или его контекст (текущие значения
счетчика команд, позиции стека, регистров процессора).

Второй аргумент 1, не равный нулю (т.е. равный TRUE) означает, что
в переменную aa будет сохраняться также и СМ (маска блокированных сигналов
и действия, связанные со всеми сигналами) текущего процесса.

Это позволяет после прерывания вернуться к последнему запомненному
состоянию. */

sigsetjmp(aa, 1); // Запоминание текущего состояния процесса
printf("Введите имя первого файла\n");
read(0, name1, 256); // Ввод имени первого файла
/* Если во время ввода имени первого файла (во время СВ read()) происходит
прерывание, то возвращение из обработчика будет произведено в то место, где
в последний раз встретилась функция sigsetjmp(), т.е. перед функцией
printf(). Т.е. при каждом нажатии на < Ctrl+C > на экран будет выводиться
сообщение "Поступил сигнал", а на выходе из прерывания будет повторен
запрос на ввод имени первого файла до тех пор, пока пользователь не введет
его.

Функция sigsetjmp() в этом месте программы будет возвращать единичку,
которая стоит во втором аргументе у siglongjmp(). Вторая функция
sigsetjmp() будет возвращать ноль. */

sigsetjmp(aa, 1); // Запоминание текущего состояния процесса
printf("Введите имя второго файла\n");
read(0, name2, 256); // Ввод имени второго файла
/* Если во время ввода имени второго файла (во время СВ read()) происходит
прерывание, то возвращение из обработчика будет произведено в то место, где
в последний (уже во второй) раз встретилась функция sigsetjmp(), т.е. перед
функцией printf(), предлагающей ввести имя второго файла.

Функция sigsetjmp() в этом месте программы будет возвращать единичку,
которая стоит во втором аргументе у siglongjmp(). Первая же функция

```

```

sigsetjmp() теперь будет возвращать ноль. */

// Вывод имен файлов на экран:
printf("Имя первого файла: %sИмя второго файла: %s\n", name1, name2);
}

1 #include <stdio.h>
2 #include <signal.h>
3 #include <setjmp.h>
4 #include <unistd.h>
5 sigjmp_buf aa;
6 void pr()
7 {
8     printf("Поступил сигнал\n");
9     siglongjmp(aa, 1);
10}
11
12
13 void main()
14 {
15     char name1[256], name2[256];
16     struct sigaction bb;
17     bb.sa_handler = pr;
18     sigprocmask(0, 0, &bb.sa_mask);
19     bb.sa_flags = 0;
20     sigaction(SIGINT, &bb, 0);
21     sigsetjmp(aa, 1);
22     printf("Введите имя первого файла\n");
23     read(0, name1, 256);
24     sigsetjmp(aa, 1);
25     printf("Введите имя второго файла\n");
26     read(0, name2, 256);
27     printf("Имя первого файла: %sИмя второго файла: %s\n", name1, name2);
28 }
29

```

Утилиты, используемые для перехвата сигналов

Помимо СВ **signal()** и **sigaction()**, которые позволяют перехватывать у ОС обработку сигнала, в программах, написанных на командном языке (shell), для перехвата сигналов можно использовать встроенную в интерпретатор утилиту **trap**. Утилита **trap** перехватывает у ОС обработку сигналов и, при получении сигнала, выполняет команду или последовательность команд. Т.е. **trap** в командном языке заменяет собой СВ **signal()** и **sigaction()**.

Подробнее с утилитой **trap** можно познакомиться в «15.4 Обработка сигналов в shell. Конструкция trap».

11. Примеры многозадачных процедурно-программных реализаций с обработкой прерываний в ОС UNIX

11.1 Постановка задачи. Подходы к реализации

Постановка задачи

Написать программу определения типа файла, указанного при запросе. Предусмотреть обработку сигнала прерывания от клавиатуры. При поступлении 3-х сигналов прерывания вывести все файлы текущего каталога, написанные на языке программирования С (или С++).

Подходы к реализации

Как известно из «4.3-4.4 Файлы в файловой системе UNIX-подобных ОС», в UNIX-подобных ОС **тип файла можно определить с помощью утилит:**

- **file <имя_файла>** — утилита, не только определяет стандартные типы файлов, но и распознает их форматы. Для утилиты file придется использовать достаточно сложный метод обработки.

```
ubuntu@ubuntu:~$ file test
test: directory
ubuntu@ubuntu:~$
```

- **ls -la <имя_файла>** (первый символ в выводе) — здесь подойдет достаточно простой метод обработки. Именно он и будет реализован в нашей программе.

Рассмотрим, за что отвечает каждый флаг по отдельности:

- l — выводить список с подробной информацией о файлах — владелец, группа, права доступа, время последнего обновления, размер и др.;

-a — отображать все файлы, включая скрытые, имена которых начинаются с точки.

Для всех файлов текущей директории:

```
ubuntu@ubuntu:~/dim/one$ ls -la
total 40
drwxrwxr-x 3 ubuntu ubuntu 4096 Mar 17 17:21 .
drwxrwxr-x 3 ubuntu ubuntu 4096 Mar 16 18:52 ..
-rwxrwxr-x 1 ubuntu ubuntu 8932 Mar 16 18:56 a.out
prw-rw-r-- 1 ubuntu ubuntu 0 Mar 17 17:21 fifo_file
-rw-rw-r-- 1 ubuntu ubuntu 95 Mar 16 18:54 main.cpp
-rwxrwxr-x 1 ubuntu ubuntu 8932 Mar 16 18:55 main.o
lrwxrwxrwx 1 ubuntu ubuntu 8 Mar 17 17:20 softlink -> main.cpp
drwxrwxr-x 2 ubuntu ubuntu 4096 Mar 17 17:19 test
```

Тип файла в выводе утилиты ls определяется по первому символу соответствующей записи.

Напомним:

1. Обычные (регулярные), первый символ в выводе для ls — символ тире (-).
2. Каталоги (d).
3. FIFO (каналы) (p).
4. Специальные (b — блок-ориентированные (устройства типа дисковых накопителей, информация в которых передается блоками), c — байт-ориентированные (устройства типа клавиатуры, информация с которых передается в виде байтов)).
5. Гнезда (s) (т.е сокеты), это файлы, которые используются для обмена информации по сети.
6. Символьная ссылка на файл (l).

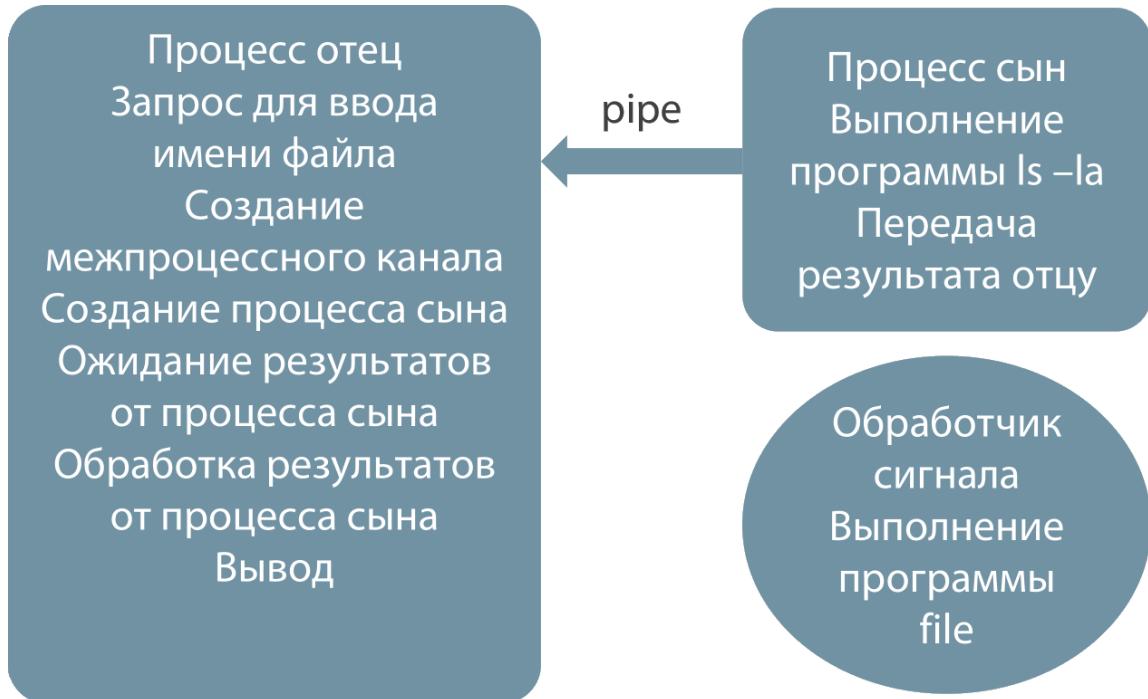
Примечание: жесткую ссылку отдельным типом файла называть нельзя, т.к. жесткая ссылка — это обычный (регулярный) файл.

Идея работы основной программы (не включая перехватчик):

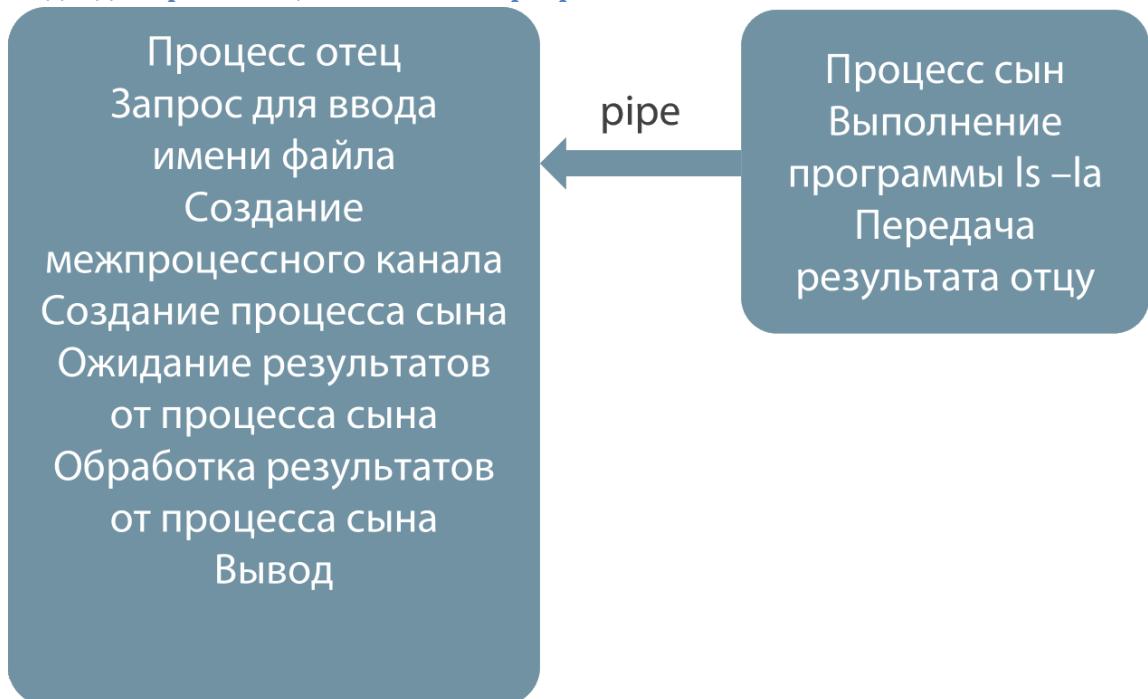
1. Ввод имени файла;
2. Создание межпроцессного канала;
3. Создание процесса-потомка;
4. Поручить ему выполнить утилиту ls с ключами -la;

5. Результат работы этой программы передать родительскому процессу по межпроцессному каналу;
6. Родительский процесс должен обработать запись, соответствующую введенному имени файла, и по номеру первого символа в этой записи определить тип файла и вывести его на экран.

Схематичное отображение этой идеи (прямоугольниками обозначены процессы, овалом обозначена функция обработчика):



Подходы к реализации. Основная программа



Вначале пишут основную программу (в данном случае — реализация процесса-отца и процесса-сына) и добиваются того, чтобы она работала безошибочно. После этого пишется функция обработчика сигнала и указание перехвата сигнала с помощью этой функции.

11.2 Пример реализации

Приступим к реализации основной программы без перехвата сигнала, состоящей из процесса-отца и процесса-сына, согласно последней схеме. Эта программа работает так, как задумано, и она удачно завершится, если во время выполнения не будет поступать никаких прерываний.

lect112-1.c:

```
// Код программы без перехвата сигнала
// Заголовочные файлы:
#include <stdio.h> // Для поддержания стандартного ввода-вывода
#include <unistd.h> /* Для возможности мобильного переноса с одной вычислительной
                     платформы на другую */
#include <stdlib.h> // Для библиотечных функций memset() и exit()
#include <string.h> /* Содержит прототипы функций работы со строками string(),
                     strcasecmp(), strcat(), strcpy(), memset(), bzero() и т.д.
                     */
#include <sys/wait.h> /* Для СВ wait(), позволяющего процессу-родителю дождаться
                         момента, когда процесс-потомок выполнит работу и пришлет
                         результаты своей работы */

/* Глобальная переменная count, видимая всем функциям данной программы,
   выполняет роль счетчика сигналов: */
int count = 0;

void main()
{
    // Выделение памяти под переменные:
    int p[2]; /* Для хранения пользовательских файловых дескрипторов межпроцессного
                канала, открываемого СВ pipe(). */
    /* Максимальная длина имени файла в UNIX-подобных ОС - 256 байт, включая
       завершающий нулевой символ, который показывает конец имени. Выделение памяти
       под символьные массивы для хранения имени файла name и буфера ввода-вывода
       buf: */
    char name[256], buf[80];

    /* Поскольку могут быть предоставлены грязные страницы памяти, то прежде, чем
       использовать выделенную память, желательно ее очистить с помощью функции
       memset() или bzero(), занеся нулевые значения в область памяти: */
    memset(name, 0, 256);
    memset(buf, 0, 80);
    /* Для повторения запроса на ввод от пользователя в случаях, когда предыдущие
       введенные пользователем значения не прошли проверку, а также когда получен
       сигнал во время ввода (пользователь ввел часть информации, но вдруг поступил
       сигнал), можно было бы применить библиотечные функции безусловного перехода
       sigsetjmp() и siglongjmp(). Однако в данном случае выбран бесконечный цикл,
       ждущий ввода имени существующего файла. Если во время ввода поступит сигнал,
       то бесконечный цикл обеспечит повторение запроса на ввод от пользователя.
       Также, пока не будет введено имя существующего файла, запрос на ввод будет
       повторяться снова и снова: */
    while (1)
    {

```

```

printf("Введите имя файла: ");
// Функция scanf() считывает с клавиатуры цепочку символов до пробела.
scanf("%s", name); // Ввод имени файла, не содержащего пробелы
/* Альтернатива 1:
   Если в именах файла предполагается использовать пробелы, то функцию
   scanf() можно заменить на СВ read(), который прочтет цепочку заданной
   длины из любых символов:
   Сброс всех буферов для того, чтобы запрос на ввод имени файла, выведенный
   через функцию printf(), успел появиться до чтения имени файла через
   СВ read(): */
// fflush(NULL);
// read(0, name, 256); // Ввод имени файла, содержащего пробелы
/* Однако, если пользователь введет слишком длинную цепочку символов,
   оказавшихся неверным именем файла, то в следующей итерации СВ read()
   продолжит считывать оставшиеся символы как новый ввод пользователя.
   Это делает использование этой функции неудобным. */
/* Альтернативы 2 и 3:
   Если в именах файла предполагается использовать пробелы, то удобнее
   применить функцию gets(), считающую строку из любых символов из
   стандартного ввода до символа перевода строки '\n' (или до конца файла)
   и записывающую ее в name: */
// gets(name); // Ввод имени файла, содержащего пробелы
/* Однако функция gets() не позволяет определить размер приемного буфера
   name, поэтому использовать ее не следует. Если входная строка окажется
   длиннее буфера, то это приведет к его переполнению и порче данных,
   которые находятся в памяти сразу после буфера.
   Поэтому в данном случае целесообразнее использовать функцию fgets(): */
// fgets(name, 256, stdin); // Ввод имени файла, содержащего пробелы

/* Посредством СВ access() проверяется, существует ли данный файл в текущей
   директории. Константа F_OK для СВ access() означает проверку на то,
   существует ли файл? Для того, чтобы проверить, разрешен ли к файлу доступ
   на чтение, в СВ access() нужно задать константу R_OK.
   Если введено корректное имя файла и этот файл существует, */
if (!access(name, F_OK))
    // то бесконечный цикл прерывается
    break;
/* ...В противном случае файл не существует, тогда бесконечный цикл
   продолжится, и запрос на ввод имени файла отправится снова.
   Если прерывание поступит во время ввода внутри этого цикла, то корректное
   имя файла не будет сформировано и снова произойдет возврат к запросу на
   ввод имени файла. */
} // Бесконечный цикл while(1)

```

/* ТПДОФ процесса:

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |

После ввода имени существующего файла СВ pipe() создает межпроцессный канал и открывает два пользовательских дескриптора файлов - p[0] с доступом к каналу на чтение и p[1] с доступом на запись, которые будут записаны в ТПДОФ (таблицу пользовательских дескрипторов открытых файлов) процесса под новыми номерами 3 и 4, соответственно:

ТПДОФ процесса после СВ pipe():

| | | | |
|------|---|-----------------------|-----|
| | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[0] | 3 | pipe _r | ... |
| p[1] | 4 | pipe _w | ... |
| | 5 | ... | ... |

*/

pipe(p);

/* После создания межпроцессного канала процесс с помощью СВ fork() разделяется на две идентичные копии, которые продолжают выполняться как два независимых процесса:

1) Процесс-потомок получает от СВ fork() код ответа 0.

2) Процесс-родитель - идентификатор, под которым запущен процесс-потомок.

В случае аварийного завершения СВ fork() вернет -1. */

if (fork() == 0)

{ /* Ветка для потомка после успешного вызова fork():

Сын

| | | | |
|------|---|-----------------------|-----|
| | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[0] | 3 | pipe _r | ... |
| p[1] | 4 | pipe _w | ... |
| | 5 | ... | ... |

Из ТПДОФ процесса-потомка удаляется запись с номером p[0] = 3, т.е.

закрывается доступ к межпроцессному каналу на чтение: */

close(p[0]);

/* Теперь потомок может только передавать информацию родителю...

Закрытие дескриптора файла стандартного вывода STDOUT_FILENO = 1

(из ТПДОФ процесса-потомка удаляется запись с номером 1):

Сын

| | | | |
|------|---|-----------------------|-----|
| | 0 | /dev/tty _r | ... |
| | 1 | | |
| | 2 | /dev/tty _w | ... |
| | 3 | | |
| p[1] | 4 | pipe _w | ... |
| | 5 | ... | ... |

*/

close(1);

/* Отождествление стандартного вывода с файловым дескриптором канала, предназначенный для записи. Для этого посредством СВ dup() выполняется

копирование дескриптора канала, предназначенного для записи (запись с номером $p[1] = 4$), в первую (наименьшую) свободную запись в ТПДОФ процесса (начиная от 0), т.е. в `STDOUT_FILENO = 1`, поскольку `STDIN_FILENO = 0` занят, а дескриптор файла стандартного вывода с номером 1 был предварительно закрыт. */

```
dup(p[1]);
```

/* Осуществлено перенаправление вывода. Вместо файла стандартного вывода, которым является монитор, установлен межпроцессный канал, который открыт на запись, поэтому результат выполнения любой стандартной инструкции, функции или утилиты будет выводиться не на экран монитора, а в межпроцессный канал.

Сын

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | pipe _w | |
| 2 | /dev/tty _w | ... |
| 3 | | |
| 4 | pipe _w | ... |
| 5 | ... | ... |

Выполнение утилиты `ls` с ключом `-la`, которая выводит список с подробной информацией обо всех файлах (владелец, группа, права доступа, время последнего обновления, размер и др.) текущего каталога, включая те файлы, имена которых начинаются с точки (скрытые). Результат выполнения этой инструкции вместо монитора будет выведен в межпроцессный канал.

Утилита `ls` выполняется с помощью СВ `execl()`, аргументы которому задаются в виде списка переменной длины, который должен заканчиваться значением `NULL`, чтобы СВ `execl()` смог определить, какой именно аргумент является последним. Первый аргумент СВ `execl()` - это полное имя программы `ls`, которая находится в папке `bin`, затем идет нулевой внешний аргумент (собственно, само имя программы `ls`), затем идут флаги `-la` и `-d` команды `ls`, затем в качестве второго внешнего аргумента используется имя файла, которое ввел пользователь. */

```
execl("/bin/ls", "ls", "-la", "-d", name, NULL);
```

/* СВ `execl()` убивает процесс, в рамках которого он был запущен, т.е. вызывает новую программу вместо уже выполняющейся без возврата в вызывающую программу. Выполнив новый процесс "`ls -la -d ИМЯФАЙЛА`", он запишет результат его выполнения в межпроцессный канал вместо монитора.

```
*/
```

```
}
```

```
else
```

```
{ /* Ветка для родителя после успешного вызова fork(): */
```

/* Родитель приостанавливает свою работу и ждет завершения процесса-потомка, вызвав СВ `wait()`. Аргумент 0 означает, что причина гибели процесса-потомка (например, по СВ `exit()` или по сигналу) нам не интересна). Поскольку процесс-потомок был всего один, то возвращаемое значение СВ `wait()` также неважно: */

```
wait(0);
```

```
/*
```

Отец

| | | | |
|------|---|-----------------------|-----|
| | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[0] | 3 | pipe _r | ... |
| p[1] | 4 | pipe _w | ... |
| | 5 | ... | ... |

После завершения процесса-потомка, из ТПДОФ процесса-родителя удаляется запись с номером p[1] = 4, т.е. закрывается доступ к межпроцессному каналу на запись:

Отец

| | | | |
|------|---|-----------------------|-----|
| | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[0] | 3 | pipe _r | ... |
| | 4 | | |
| | 5 | ... | ... |

```
/*
close(p[1]);
/* Теперь родитель может только читать информацию от потомка...
Бесконечный цикл, обрабатывающий результаты работы процесса-потомка: */
while (1)
{   /* Считывание данных из межпроцессного канала в массив buf по 80 байтам.
    Если прочитан хотя бы один байт, то */
    if (read(p[0], buf, 80) != 0)
    {   /* считанные данные обрабатываются - определяется тип файла
        по первому символу (первому байту) в массиве buf: */
        if (buf[0] == '-')
            printf("файл - обычный\n");
        if (buf[0] == 'd')
            printf("файл директория\n");
        if (buf[0] == 'p')
            printf("файл канал\n");
        if (buf[0] == 'c')
            printf("файл специальный байт-ориентированный\n");
        if (buf[0] == 'b')
            printf("файл специальный блок-ориентированный\n");
        if (buf[0] == 's')
            printf("файл гнездо\n");
        if (buf[0] == 'l')
            printf("файл символьная ссылка\n");
        /* Очистка символьного массива buf с помощью функции memset() путем
           занесения нулевых значений в область памяти: */
        memset(buf, 0, 80);
    }
    /* Когда СВ read() вернет 0, будет достигнут конец файла межпроцессного
       канала p[0]: */
```

```

    else
        break;
    /* Завершается выполнение основной программы. Библиотечная функция
       exit() сбрасывает буферы ввода-вывода и завершает процесс, закрывая
       все открытые в процессе файлы; код 0 означает правильное завершение
       процесса-родителя: */
    exit(0);
} // Бесконечный цикл while(1)
}
}

```

Код программы без перехвата сигнала

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/wait.h>
6 void main()
7 {int fd; int p[2]; char name[256], buf[80];
8 memset( name, 0, 256 ); memset( buf, 0, 80 );
9 while(1)
10 {
11 printf( "Введите имя файла:" );
12 scanf( "%s", name );
13 if(!access(name, F_OK)) break;
14 }
15 pipe(p);
16 if( fork() == 0 )
17 {
18 close(p[0]);
19 close( 1 );
20 dup( p[1] );
21 execl( "/bin/ls", "ls", "-la", "-d", name, NULL );
22 } else {
23 wait(0);
24 close(p[1]);
25 ▼ while(1){
26     if(read(p[0], buf, 80) != 0)
27     {if( buf[0] == '-' ) printf( "файл - обычный\n");
28      if( buf[0] == 'd' ) printf( "файл директория\n");
29      if( buf[0] == 'p' ) printf( "файл канал\n");
30      if( buf[0] == 'c' ) printf( "файл специальный байт-ориентированный\n");
31      if( buf[0] == 'b' ) printf( "файл специальный блок-ориентированный\n");
32      if( buf[0] == 's' ) printf( "файл гнездо\n");
33      if( buf[0] == 'l' ) printf( "файл символьная ссылка\n");
34      memset( buf, 0, 80 ); } else break; exit( 0 );
35 } } }

```

Примечание: на скриншоте присутствует переменная fd, которая в программе затем никак не используется.

Далее нужно разработать обработчик сигналов и включить его в код основной программы.

11.3 Пример реализации (продолжение)

Обработчик прерывания. Подходы к решению

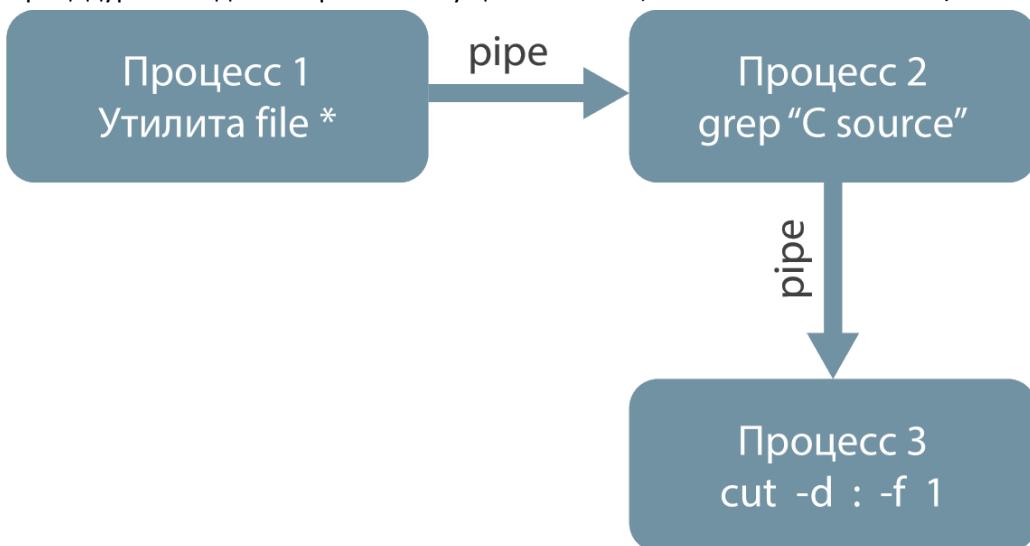
Согласно заданию, нужно предусмотреть обработку сигнала прерывания от клавиатуры. При поступлении 3-х сигналов прерывания вывести все файлы текущего каталога, написанные на языке программирования С (или C++). В качестве прерывания будет использовано нажатие на клавиатуре комбинации клавиш < Ctrl+C > (сигнал прерывания SIGINT, код 2).

Для начала определим, с помощью каких системных средств можно получить эту информацию.

Во-первых, нужно определить, какие файлы в текущем каталоге написаны на языке C/C++. Для этого подойдут две утилиты:

- **file** (1-ый вариант, работает намного быстрее, он и будет реализован) — определяет тип файла и анализирует его содержимое. Она анализирует первые записи файла и на основе этого анализа выводит свою характеристику указанного файла. Она может показать тип файла — текстовый (в какой кодировке и на каком языке программирования (Pascal, C/C++, командный язык) он написан) или двоичный файл (исполняемый файл или картинка и т.д.).
- **find** (2-ой вариант) — поиск файла в любых каталогах ФС UNIX с указанными параметрами. Имеет неограниченное количество внешних аргументов, т.е. характеристики поиска можно задавать очень большим количеством внешних аргументов.

Процедура вывода всех файлов текущего каталога, написанных на языке C/C++.



Утилита file определяет тип файла и анализирует его содержимое. Результаты утилиты file через межпроцессный канал передаются другому процессу. Для связи между процессами выбран именно межпроцессный канал, потому что это файл в ОП, а через ОП — наиболее быстрый способ передачи информации. Результат выполнения команды `file *`, которая в каждом из файлов текущего каталога анализирует первые 10 записей и выводит имя файла, затем символ двоеточия, затем тип этого файла и другую информацию:

```

ubuntu@ubuntu:~/dim/one$ ubuntu@ubuntu:~/dim/one$ file *
a.c:          C source, ASCII text
a.c.save:     C source, ASCII text
a.c.save.1:   C source, ASCII text
a.out:        ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/l
d-linux-armhf.so.3, BuildID[sha1]=74fe37fa8033601c1a906702b69db47a1a1ccfe9, for GNU/Linux 3.2.0, not stripped
fifo_file:   fifo (named pipe)
lab.c:        C source, ASCII text
main.cpp:     C source, ASCII text
main.o:       ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/l
d-linux-armhf.so.3, BuildID[sha1]=f2d0c80d33ca940a49544404d0c09c90c69dc614, for GNU/Linux 3.2.0, not stripped
old.c:        C source, UTF-8 Unicode text
softlink:     symbolic link to main.cpp
sone.c:       C source, UTF-8 Unicode text
test:         directory
test.c:       C source, UTF-8 Unicode text
test.c.save:  C source, UTF-8 Unicode text
  
```

Результат команды `file` передается через межпроцессный канал во второй процесс — на вход утилиты `grep`, которая выводит только те записи, которые содержат шаблон "C source". Вместо "C source" может быть "C program" — в зависимости от ОС. Результат выполнения команды `file * | grep "C source"`:

```
ubuntu@ubuntu:~/dim/one$ file * | grep "C source"
a.c:          C source, ASCII text
a.c.save:     C source, ASCII text
a.c.save.1:   C source, ASCII text
end.txt:      C source, UTF-8 Unicode text
end.txt.save: C source, UTF-8 Unicode text
lab.c:        C source, ASCII text
labFULL.c:   C source, UTF-8 Unicode text
main.cpp:    C source, ASCII text
old.c:       C source, UTF-8 Unicode text
sone.c:      C source, UTF-8 Unicode text
test.c:      C source, UTF-8 Unicode text
test.c.save: C source, UTF-8 Unicode text
```

Затем полученный набор записей, содержащих искомый шаблон передается через межпроцессный канал в третий процесс — на вход утилиты cut, которая вырезает первый столбец до символа разделения ":".

Результат выполнения конвейера из трех утилит выведет имена всех файлов текущего каталога, написанных на языке программирования С/С++. Результат выполнения команды `file * | grep "C source" | cut -d : -f 1`

```
ubuntu@ubuntu:~/dim/one$ file * | grep "C source" | cut -d : -f 1
a.c
a.c.save
a.c.save.1
end.txt
end.txt.save
lab.c
labFULL.c
main.cpp
old.c
sone.c
test.c
test.c.save
```

Обработчик прерываний. Реализация

Курсивом выделены уже реализованные участки кода основной программы.

`lect113-1.c:`

```
// Заголовочные файлы
/* Глобальная переменная count, видимая всем функциям данной программы,
выполняет роль счетчика сигналов. */

/* Функция обработки сигнала handle() должна быть описана или объявлена перед
использованием СВ signal() или sigaction().
Функция обработки сигнала запускает два процесса-потомка и открывает два
межпроцессных канала. Для выполнения задачи достаточно трех процессов:
1. первый потомок выполняет утилиту file;
2. второй потомок - grep;
3. третий (родитель) - cut. */
void handle() // Описание функции обработчика сигнала
{
    /* Выделение памяти под переменные для хранения пользовательских файловых
дескрипторов двух межпроцессных каналов, открываемых СВ pipe(). Канал p
будет использован для связи с первым процессом-потомком, канал p1 - для
связи со вторым потомком: */
    int p[2], p1[2];
    /* Счетчик сигналов прерывания увеличивается при поступлении каждого сигнала: */
    count++;
    /* При поступлении 3-х сигналов прерывания нужно вывести имена всех файлов
текущего каталога, написанных на языке программирования С или С++.
Эту информацию можно получить, реализовав конвейерное выполнение трех утилит
file (или find, которая работает медленнее), grep и cut:
"file * | grep "C source" | cut -d : -f 1"
В некоторых ОС утилита file может выводить "C program", тогда в качестве
аргумента утилиты grep вместо шаблона "C source" нужно использовать
```

```
"C program". */
if (count == 3)
{
```

/* ТПДОФ процесса:

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| | | |

Первый межпроцессный канал будет использован для общения с первым потомком, который будет создан для выполнения утилиты file.

СВ pipe() создает первый межпроцессный канал и открывает два пользовательских дескриптора файлов - p[0] с доступом к каналу на чтение и p[1] с доступом на запись, которые будут записаны в ТПДОФ (таблицу пользовательских дескрипторов открытых файлов) процесса под новыми номерами 3 и 4, соответственно:

ТПДОФ процесса после СВ pipe():

| | | | |
|------|---|-----------------------|-----|
| p[0] | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[1] | 3 | pipe _r | ... |
| | 4 | pipe _w | ... |
| | 5 | ... | ... |

*/

pipe(p);

/* После создания первого межпроцессного канала процесс с помощью СВ fork() разделяется на две идентичные копии, которые продолжают выполняться как два независимых процесса:

1) Процесс-потомок получает от СВ fork() код ответа 0.

2) Процесс-родитель - идентификатор, под которым запущен процесс-потомок.

В случае аварийного завершения СВ fork() вернет -1.

Создание первого процесса-потомка для выполнения утилиты file: */

if (fork() == 0)

{ /* Ветка для первого потомка (выполнит утилиту file) после успешного вызова fork():

Сын

| | | | |
|------|---|-----------------------|-----|
| p[0] | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[1] | 3 | pipe _r | ... |
| | 4 | pipe _w | ... |
| | 5 | ... | ... |

Из ТПДОФ процесса-потомка удаляется запись с номером p[0] = 3, т.е.

закрывается доступ к межпроцессному каналу на чтение:

Сын

```

0 /dev/ttyr ...
1 /dev/ttyw ...
2 /dev/ttyw ...
...
p[1] 4 pipew ...
5 ... ...
*/
close(p[0]);
/* Отождествление стандартного вывода с файловым дескриптором первого
канала, предназначенным для записи. Для этого посредством СВ dup2()
выполняется принудительное копирование дескриптора канала,
предназначенного для записи (запись с номером p[1] = 4), в запись
STDOUT_FILENO = 1 из ТПДОФ. Поскольку дескриптор STDOUT_FILENO = 1
перед вызовом dup2() был открыт, то он предварительно закрывается:

```

Сын

```

0 /dev/ttyr ...
1 pipew ...
2 /dev/ttyw ...
...
p[1] 4 pipew ...
5 ... ...
*/
dup2(p[1], 1);
/* Осуществлено перенаправление вывода. Вместо файла стандартного
вывода, которым является монитор, установлен первый межпроцессный
канал, который открыт на запись, поэтому результат выполнения любой
стандартной инструкции, функции или утилиты будет выводиться не
на экран монитора, а в первый межпроцессный канал.
СВ system() позволяет вызывающей программе выполнить произвольную
консольную команду или вызвать другую программу, не выходя из
контекста вызывающего процесса. СВ system() выполняет команду для
утилиты "file *", результат работы этой утилиты поступает в
первый межпроцессный канал, а не выводится на монитор:

```

Сын

Результат file *



```

0 /dev/ttyr ...
1 pipew ...
2 /dev/ttyw ...
...
p[1] 4 pipew ...
5 ... ...
*/
system("file *");
/* Первый процесс-потомок закрывается с помощью СВ _exit(0), который
закрывает все открытые в процессе файлы. Код 0 означает правильное

```

```

    завершение процесса-потомка: */
    _exit(0);
} // Ветка для первого потомка
else
{
    /* Ветка для родителя после успешного вызова fork(). Родитель создаст
       второго потомка, который выполнит утилиту grep, а затем сам родитель
       выполнит утилиту cut.

       Вызвав СВ wait(), родитель приостанавливает свою работу и ждет
       завершения первого процесса-потомка, выполнившего утилиту file.

       Аргумент 0 означает, что причина гибели процесса-потомка (например,
       по СВ exit() или по сигналу) нам не интересна). Поскольку процесс-
       потомок к данному моменту был всего один, то возвращаемое значение
       СВ wait() также неважно: */

    wait(0);
    /* Первый канал p в родительском процессе все еще существует, т.к.
       межпроцессный канал уничтожается только тогда, когда от него
       отсоединяются все процессы и, хотя первый потомок завершен, но
       родительский процесс все еще работает.

```

Отец

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | pipe _w | ... |
| 5 | ... | ... |

Результат file *

→ p[0]
p[1]

Второй межпроцессный канал будет использован для общения со вторым потомком, который будет создан для выполнения утилиты grep.

После завершения процесса первого потомка, процесс-родитель с помощью СВ pipe() создает второй межпроцессный канал и открывает два пользовательских дескриптора файлов - p1[0] с доступом к каналу на чтение и p1[1] с доступом на запись, которые будут записаны в ТПДОФ процесса под новыми номерами 5 и 6, соответственно:

Отец

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | pipe _w | ... |
| 5 | 1pipe _r | ... |
| 6 | 1pipe _w | ... |
| 7 | | |
| 8 | | |

*/

pipe(p1);

/* Сторона первого канала, открытая для записи, больше не нужна, т.к.

результат работы первого потомка (результат выполнения утилиты file) уже хранится в файле канала с номером пользовательского дескриптора $p[0]$, открытый на чтение.

Из ТПДОФ процесса-родителя удаляется запись с номером $p[1] = 4$, т.е. закрывается доступ на запись к первому межпроцессному каналу, использованному для общения с первым потомком:

Отец

| | | |
|---|-----------------------|-----|
| | | |
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | | |
| 5 | 1pipe _r | ... |
| 6 | 1pipe _w | ... |
| 7 | | |
| 8 | | |

*/

close(p[1]);

/* После создания второго межпроцессного канала процесс с помощью СВ fork() разделяется на две идентичные копии, которые продолжают выполняться как два независимых процесса:

- 1) Процесс-потомок получает от СВ fork() код ответа 0.
- 2) Процесс-родитель - идентификатор, под которым запущен процесс-потомок.

В случае аварийного завершения СВ fork() вернет -1.

Создание второго процесса-потомка для выполнения утилиты grep: */

```
if (fork() == 0)
{ /* Ветка для второго потомка (выполнит утилиту grep) после успешного
    вызова fork():
```

Сын 2

| | | |
|---|-----------------------|-----|
| | | |
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | | |
| 5 | 1pipe _r | ... |
| 6 | 1pipe _w | ... |
| 7 | | |
| 8 | | |

Из ТПДОФ второго потомка удаляется запись с номером $p1[0] = 5$, т.е. закрывается доступ ко второму межпроцессному каналу на чтение:

Сын 2

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | | |
| 5 | | |
| 6 | 1pipe _w | ... |
| 7 | | |
| 8 | | |

```
*/
close(p1[0]);
/* Отождествление стандартного вывода с файловым дескриптором
второго канала, предназначенным для записи. Для этого посредством
СВ dup2() выполняется принудительное копирование дескриптора
канала, предназначенного для записи (запись с номером p1[1] = 6),
в запись STDOUT_FILENO = 1 из ТПДОФ. Поскольку дескриптор
STDOUT_FILENO = 1 перед вызовом dup2() был открыт, то он
предварительно закрывается:
```

Сын 2

| | | |
|---|-----------------------|-----|
| 0 | /dev/tty _r | ... |
| 1 | 1pipe _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | | |
| 5 | | |
| 6 | 1pipe _w | ... |
| 7 | | |
| 8 | | |



```
*/
dup2(p1[1], 1);
/* Осуществлено перенаправление вывода. Вместо файла стандартного
вывода, которым является монитор, установлен второй межпроцессный
канал, который открыт на запись, поэтому результат выполнения
любой стандартной инструкции, функции или утилиты будет
выводиться не на экран монитора, а во второй межпроцессный канал.
Второй потомок должен считать данные от утилиты file и
использовать их в качестве входных для выполнения утилиты grep.
Поэтому производится отождествление стандартного ввода с файловым
дескриптором первого канала, предназначенным для чтения.
Для этого посредством СВ dup2() выполняется принудительное
копирование дескриптора первого канала, предназначенного для
чтения (запись с номером p[0] = 3), в запись STDIN_FILENO = 0 из
ТПДОФ. Поскольку дескриптор STDIN_FILENO = 0 перед вызовом dup2()
был открыт, то он предварительно закрывается:
```

Сын 2

| | | |
|---|-----------------------|-----|
| 0 | pipe _r | ... |
| 1 | 1pipe _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | | |
| 5 | | |
| 6 | 1pipe _w | ... |
| 7 | | |
| 8 | | |

```
*/
dup2(p[0], 0);
/* Осуществлено перенаправление ввода. Вместо файла стандартного
ввода, которым является клавиатура, установлен первый
межпроцессный канал, который открыт на чтение. Теперь данные,
полученные от утилиты file, которые содержатся в первом
межпроцессном канале, будут поступать из стандартного ввода
в утилиту grep.
В результате выполненных перенаправлений ввода-вывода, утилита
grep считает данные из первого канала p[0] и выведет данные во
второй канал p1[1].
```

Сын 2 *grep*

| | | | | |
|--------|---|---|-----------------------|-----|
| file * | → | 0 | pipe _r | ... |
| | | 1 | 1pipe _w | ... |
| | | 2 | /dev/tty _w | ... |
| | | 3 | pipe _r | ... |
| | | 4 | | |
| | | 5 | | |
| | | 6 | 1pipe _w | ... |
| | | 7 | | |
| | | 8 | | |

СВ system() позволяет вызывающей программе выполнить произвольную консольную команду или вызвать другую программу, не выходя из контекста вызывающего процесса. СВ system() выполняет команду для утилиты "grep "C source"" , результат работы этой утилиты поступает во второй межпроцессный канал, а не выводится на монитор: */

```
system("grep \"C source\"");
/* Шаблон "C source" для утилиты grep состоит из двух слов. Шаблон,
состоящий из двух слов и более необходимо обрамлять кавычками.
Левый слеш используется для экранирования символа двойных
кавычек, чтобы внутри одних кавычек сохранились вторые, вложенные
кавычки.
Второй процесс-потомок закрывается с помощью СВ _exit(1), который
закрывает все открытые в процессе файлы. Код 1 означает
```

```

    правильное завершение процесса-потомка: */
    _exit(1);
}
else
{
    /* Ветка для родителя после успешного вызова второго СВ fork().
       Продолжение кода родительского процесса, в котором он выполнит
       утилиту cut. */
    /* Вызвав СВ wait(), родитель приостанавливает свою работу и ждет
       завершения второго процесса-потомка, выполнившего утилиту grep.
       Аргумент 0 означает, что причина гибели процесса-потомка
       (например, по СВ exit() или по сигналу) нам не интересна).
       Поскольку первый процесс-потомок к данному моменту был уже
       завершен и остался только второй потомок, то возвращаемое
       значение СВ wait() также неважно: */
    wait(0);
/*

```

Отец

| | | | |
|-------|---|-----------------------|-----|
| | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[0] | 3 | pipe _r | ... |
| | 4 | | |
| p1[0] | 5 | 1pipe _r | ... |
| p1[1] | 6 | 1pipe _w | ... |
| | 7 | | |
| | 8 | | |

←

Результат
grep ...

К данному моменту завершились оба процесса-потомка.

Из ТПДОФ родительского процесса удаляется запись с номером p1[1] = 6, т.е. закрывается доступ на запись ко второму межпроцессному каналу, использованному для общения со вторым потомком:

Отец

| | | | |
|-------|---|-----------------------|-----|
| | 0 | /dev/tty _r | ... |
| | 1 | /dev/tty _w | ... |
| | 2 | /dev/tty _w | ... |
| p[0] | 3 | pipe _r | ... |
| | 4 | | |
| p1[0] | 5 | 1pipe _r | ... |
| | 6 | | |
| | 7 | | |
| | 8 | | |

←

Результат
grep ...

*/

close(p1[1]);

/* Родитель должен считать данные от утилиты grep и использовать их

в качестве входных для выполнения утилиты `cut`. Поэтому производится отождествление стандартного ввода с файловым дескриптором второго канала, предназначенным для чтения. Для этого посредством СВ `dup2()` выполняется принудительное копирование дескриптора второго канала, предназначенного для чтения (запись с номером `p1[0] = 5`), в запись `STDIN_FILENO = 0` из ТПДОФ. Поскольку дескриптор `STDIN_FILENO = 0` перед вызовом `dup2()` был открыт, то он предварительно закрывается:

Отец

| | | |
|---|-----------------------|-----|
| 0 | 1pipe _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | | |
| 5 | 1pipe _r | ... |
| 6 | | |
| 7 | | |
| 8 | | |

Результат
grep ...

```
/*
dup2(p1[0], 0);
/* Осуществлено перенаправление ввода. Вместо файла стандартного
   ввода, которым является клавиатура, установлен второй
   межпроцессный канал, который открыт на чтение. Теперь данные,
   полученные от утилиты grep, которые содержатся во втором
   межпроцессном канале, будут поступать из стандартного ввода
   в утилиту cut.
```

Отец

| | | |
|---|-----------------------|-----|
| 0 | 1pipe _r | ... |
| 1 | /dev/tty _w | ... |
| 2 | /dev/tty _w | ... |
| 3 | pipe _r | ... |
| 4 | | |
| 5 | 1pipe _r | ... |
| 6 | | |
| 7 | | |
| 8 | | |

Результат
grep ...

cut

СВ `system()` позволяет вызывающей программе выполнить произвольную консольную команду или вызвать другую программу, не выходя из контекста вызывающего процесса. СВ `system()` выполняет команду для утилиты "cut -d : -f 1", которая вырезает первое поле с именами файлов до разделителя двоеточие. В результате выполненного перенаправления ввода, утилита `cut` считает данные из второго канала `p1[0]` и выведет данные в стандартный вывод, т.е. на экран монитора: */

```
system("cut -d : -f 1");
```

```

/* Родительский процесс (т.е. вся программа) закрывается с помощью
СВ _exit(0), который закрывает все открытые в процессе файлы.
Код 0 означает правильное завершение процесса: */
_exit(0);
} // Ветка для родителя после успешного вызова второго СВ fork().
} // Ветка для родителя после успешного вызова fork().
} // При поступлении 3-х сигналов прерывания.
} // Функция обработчика сигнала.

// void main() - главная программа...

```

```

1 void handle() {
2     int p[2], p1[2];
3     pipe(p);
4     if(fork() == 0) {
5         close(p[0]);
6         dup2(p[1], 1);
7         system("file *");
8         _exit(0);
9     } else {
10        wait(0);
11        pipe(p1);
12        close(p[1]);
13        if(fork() == 0) {
14            close(p1[0]);
15            dup2(p1[1],1);
16            dup2(p[0], 0);
17            system("grep \"C source\"");
18            _exit(1);
19        } else {
20            wait(0);
21            close(p1[1]);
22            dup2(p1[0], 0);
23            system("cut -d : -f 1");
24            _exit(0);
25        } } }

```

Примечание: на скриншоте пропущены две строки и открывающая/закрывающая фигурные скобки:

```

count++; // увеличение счетчика сигналов
if (count == 3)
{ .... }

```

11.4 Пример реализации (продолжение). Часть 2

Приступим к встраиванию обработчика сигнала в основную программу. Жирным выделены новые участки кода, не рассмотренные ранее. Добавлены:

- заголовочный файл signal.h;
- функция handle();
- настройка обработчика сигнала.

lect114-1.c:

```

// Код программы со встроенным обработчиком сигнала
// Заголовочные файлы:
#include <stdio.h>      // Для поддержания стандартного ввода-вывода
#include <unistd.h>      /* Для возможности мобильного переноса с одной вычислительной
                           платформы на другую */
#include <stdlib.h>       // Для библиотечных функций memset() и exit()
#include <string.h>       /* Содержит прототипы функций работы со строками string(),
                           strcasecmp(), strcat(), strcpy(), memset(), bzero() и т.д.
                           */

```

```

#include <sys/wait.h> /* Для СВ wait(), позволяющего процессу-родителю дождаться
                      момента, когда процесс-потомок выполнит работу и пришлет
                      результаты своей работы */
#include <signal.h> /* Для работы с сигнальной маской процесса и макросами,
                      прототипами данных из функции sigaction() */

/* Глобальная переменная count, видимая всем функциям данной программы,
   выполняет роль счетчика сигналов: */
int count = 0;

/* Функция обработки сигнала handl() должна быть описана или объявлена перед
   использованием СВ signal() или sigaction().
   Функция обработки сигнала запускает два процесса-потомка и открывает два
   межпроцессных канала. Для выполнения задачи достаточно трех процессов:
   1. первый потомок выполняет утилиту file;
   2. второй потомок - grep;
   3. третий (родитель) - cut. */
void handl() // Описание функции обработчика сигнала
{
    /* Выделение памяти под переменные для хранения пользовательских файловых
       дескрипторов двух межпроцессных каналов, открываемых СВ pipe(). Канал p
       будет использован для связи с первым процессом-потомком, канал p1 - для
       связи со вторым потомком: */
    int p[2], p1[2];
    /* Счетчик сигналов прерывания увеличивается при поступлении каждого сигнала: */
    count++;
    /* При поступлении 3-х сигналов прерывания нужно вывести имена всех файлов
       текущего каталога, написанных на языке программирования С или С++.
       Эту информацию можно получить, реализовав конвейерное выполнение трех утилит
       file (или find, которая работает медленнее), grep и cut:
       "file * | grep "C source" | cut -d : -f 1"
       В некоторых ОС утилита file может выводить "C program", тогда в качестве
       аргумента утилиты grep вместо шаблона "C source" нужно использовать
       "C program". */
    if (count == 3)
    {
        /* Первый межпроцессный канал будет использован для общения с первым
           потомком, который будет создан для выполнения утилиты file.
           СВ pipe() создает первый межпроцессный канал и открывает два
           пользовательских дескриптора файлов - p[0] с доступом к каналу на чтение
           и p[1] с доступом на запись, которые будут записаны в ТПДОФ (таблицу
           пользовательских дескрипторов открытых файлов) процесса под новыми
           номерами 3 и 4, соответственно: */
        pipe(p);
        /* После создания первого межпроцессного канала процесс с помощью СВ fork()
           разделяется на две идентичные копии, которые продолжают выполняться как
           два независимых процесса:
           1) Процесс-потомок получает от СВ fork() код ответа 0.
           2) Процесс-родитель - идентификатор, под которым запущен процесс-потомок.
           В случае аварийного завершения СВ fork() вернет -1. */
    }
}

```

```

Создание первого процесса-потомка для выполнения утилиты file: */
if (fork() == 0)
{
    /* Ветка для первого потомка (выполнит утилиту file) после успешного
       вызова fork():
       Из ТПДОФ процесса-потомка удаляется запись с номером p[0] = 3, т.е.
       закрывается доступ к межпроцессному каналу на чтение: */
    close(p[0]);
    /* Отождествление стандартного вывода с файловым дескриптором первого
       канала, предназначенный для записи. Для этого посредством СВ dup2()
       выполняется принудительное копирование дескриптора канала,
       предназначенного для записи (запись с номером p[1] = 4), в запись
       STDOUT_FILENO = 1 из ТПДОФ. Поскольку дескриптор STDOUT_FILENO = 1
       перед вызовом dup2() был открыт, то он предварительно закрывается: */
    dup2(p[1], 1);
    /* Осуществлено перенаправление вывода. Вместо файла стандартного
       вывода, которым является монитор, установлен первый межпроцессный
       канал, который открыт на запись, поэтому результат выполнения любой
       стандартной инструкции, функции или утилиты будет выводиться не
       на экран монитора, а в первый межпроцессный канал.
       СВ system() позволяет вызывающей программе выполнить произвольную
       консольную команду или вызвать другую программу, не выходя из
       контекста вызывающего процесса. СВ system() выполняет команду для
       утилиты "file *", результат работы этой утилиты поступает в
       первый межпроцессный канал, а не выводится на монитор: */
    system("file *");
    /* Первый процесс-потомок закрывается с помощью СВ _exit(0), который
       закрывает все открытые в процессе файлы. Код 0 означает правильное
       завершение процесса-потомка: */
    _exit(0);
} // Ветка для первого потомка
else
{
    /* Ветка для родителя после успешного вызова fork(). Родитель создаст
       второго потомка, который выполнит утилиту grep, а затем сам родитель
       выполнит утилиту cut.
       Вызвав СВ wait(), родитель приостанавливает свою работу и ждет
       завершения первого процесса-потомка, выполнившего утилиту file.
       Аргумент 0 означает, что причина гибели процесса-потомка (например,
       по СВ exit() или по сигналу) нам не интересна). Поскольку процесс-
       потомок к данному моменту был всего один, то возвращаемое значение
       СВ wait() также неважно: */
    wait(0);
    /* Первый канал p в родительском процессе все еще существует, т.к.
       межпроцессный канал уничтожается только тогда, когда от него
       отсоединяются все процессы и, хотя первый потомок завершен, но
       родительский процесс все еще работает.
       Второй межпроцессный канал будет использован для общения со вторым
       потомком, который будет создан для выполнения утилиты grep.
       После завершения процесса первого потомка, процесс-родитель с помощью
       СВ pipe() создает второй межпроцессный канал и открывает два

```

пользовательских дескриптора файлов - $p1[0]$ с доступом к каналу на чтение и $p1[1]$ с доступом на запись, которые будут записаны в ТПДОФ процесса под новыми номерами 5 и 6, соответственно: */

```
pipe(p1);
/* Сторона первого канала, открытая для записи, больше не нужна, т.к.
результат работы первого потомка (результат выполнения утилиты file)
уже хранится в файле канала с номером пользовательского дескриптора
р[0], открытым на чтение.
Из ТПДОФ процесса-родителя удаляется запись с номером р[1] = 4, т.е.
закрывается доступ на запись к первому межпроцессному каналу,
использованному для общения с первым потомком: */
close(p[1]);
/* После создания второго межпроцессного канала процесс с помощью
СВ fork() разделяется на две идентичные копии, которые продолжают
выполняться как два независимых процесса:
1) Процесс-потомок получает от СВ fork() код ответа 0.
2) Процесс-родитель - идентификатор, под которым запущен процесс-
потомок.

В случае аварийного завершения СВ fork() вернет -1.
Создание второго процесса-потомка для выполнения утилиты grep: */
if (fork() == 0)
{
    /* Ветка для второго потомка (выполнит утилиту grep) после успешного
    вызова fork():
    Из ТПДОФ второго потомка удаляется запись с номером p1[0] = 5,
    т.е. закрывается доступ ко второму межпроцессному каналу
    на чтение: */
    close(p1[0]);
    /* Отождествление стандартного вывода с файловым дескриптором
    второго канала, предназначенным для записи. Для этого посредством
    СВ dup2() выполняется принудительное копирование дескриптора
    канала, предназначенного для записи (запись с номером p1[1] = 6),
    в запись STDOUT_FILENO = 1 из ТПДОФ. Поскольку дескриптор
    STDOUT_FILENO = 1 перед вызовом dup2() был открыт, то он
    предварительно закрывается: */
    dup2(p1[1], 1);
    /* Осуществлено перенаправление вывода. Вместо файла стандартного
    вывода, которым является монитор, установлен второй межпроцессный
    канал, который открыт на запись, поэтому результат выполнения
    любой стандартной инструкции, функции или утилиты будет
    выводиться не на экран монитора, а во второй межпроцессный канал.
    Второй потомок должен считать данные от утилиты file и
    использовать их в качестве входных для выполнения утилиты grep.
    Поэтому производится отождествление стандартного ввода с файловым
    дескриптором первого канала, предназначенным для чтения.

    Для этого посредством СВ dup2() выполняется принудительное
    копирование дескриптора первого канала, предназначенного для
    чтения (запись с номером p[0] = 3), в запись STDIN_FILENO = 0 из
    ТПДОФ. Поскольку дескриптор STDIN_FILENO = 0 перед вызовом dup2()
    был открыт, то он предварительно закрывается: */
    dup2(p[0], 0);
```

```

/* Осуществлено перенаправление ввода. Вместо файла стандартного
   ввода, которым является клавиатура, установлен первый
   межпроцессный канал, который открыт на чтение. Теперь данные,
   полученные от утилиты file, которые содержатся в первом
   межпроцессном канале, будут поступать из стандартного ввода
   в утилиту grep.
В результате выполненных перенаправлений ввода-вывода, утилита
grep считает данные из первого канала p[0] и выведет данные во
второй канал p1[1].
СВ system() позволяет вызывающей программе выполнить произвольную
консольную команду или вызвать другую программу, не выходя из
контекста вызывающего процесса. СВ system() выполняет команду для
утилиты "grep "C source""", результат работы этой утилиты
поступает во второй межпроцессный канал, а не выводится
на монитор: */
system("grep \"C source\"");
/* Шаблон "C source" для утилиты grep состоит из двух слов. Шаблон,
   состоящий из двух слов и более необходимо обрамлять кавычками.
   Левый слеш используется для экранирования символа двойных
   кавычек, чтобы внутри одних кавычек сохранились вторые, вложенные
   кавычки.
Второй процесс-потомок закрывается с помощью СВ _exit(1), который
закрывает все открытые в процессе файлы. Код 1 означает
правильное завершение процесса-потомка: */
_exit(1);
}
else
{
/* Ветка для родителя после успешного вызова второго СВ fork().
   Продолжение кода родительского процесса, в котором он выполнит
   утилиту cut. */
/* Вызвав СВ wait(), родитель приостанавливает свою работу и ждет
   завершения второго процесса-потомка, выполнившего утилиту grep.
   Аргумент 0 означает, что причина гибели процесса-потомка
   (например, по СВ exit() или по сигналу) нам не интересна).
   Поскольку первый процесс-потомок к данному моменты был уже
   завершен и остался только второй потомок, то возвращаемое
   значение СВ wait() также неважно: */
wait(0);
/* К данному моменту завершились оба процесса-потомка.
   Из ТПДОФ родительского процесса удаляется запись с номером
   p1[1] = 6, т.е. закрывается доступ на запись ко второму
   межпроцессному каналу, использованному для общения со вторым
   потомком: */
close(p1[1]);
/* Родитель должен считать данные от утилиты grep и использовать их
   в качестве входных для выполнения утилиты cut. Поэтому
   производится отождествление стандартного ввода с файловым
   дескриптором второго канала, предназначенным для чтения.
   Для этого посредством СВ dup2() выполняется принудительное

```

```

копирование дескриптора второго канала, предназначенного для
чтения (запись с номером p1[0] = 5), в запись STDIN_FILENO = 0 из
ТПДОФ. Поскольку дескриптор STDIN_FILENO = 0 перед вызовом dup2()
был открыт, то он предварительно закрывается: */
dup2(p1[0], 0);
/* Осуществлено перенаправление ввода. Вместо файла стандартного
ввода, которым является клавиатура, установлен второй
межпроцессный канал, который открыт на чтение. Теперь данные,
полученные от утилиты grep, которые содержатся во втором
межпроцессном канале, будут поступать из стандартного ввода
в утилиту cut.
СВ system() позволяет вызывающей программе выполнить произвольную
консольную команду или вызвать другую программу, не выходя из
контекста вызывающего процесса. СВ system() выполняет команду для
утилиты "cut -d : -f 1", которая вырезает первое поле с именами
файлов до разделителя двоеточие. В результате выполненного
перенаправления ввода, утилита cut считает данные из второго
канала p1[0] и выведет данные в стандартный вывод, т.е. на экран
монитора: */
system("cut -d : -f 1");
/* Родительский процесс (т.е. вся программа) закрывается с помощью
СВ _exit(0), который закрывает все открытые в процессе файлы.
Код 0 означает правильное завершение процесса: */
_exit(0);
} // Ветка для родителя после успешного вызова второго СВ fork().
} // Ветка для родителя после успешного вызова fork().
} // При поступлении 3-х сигналов прерывания.
} // Функция обработчика сигнала.

void main()
{
    // Выделение памяти под переменные:
    int p[2]; /* Для хранения пользовательских файловых дескрипторов межпроцессного
               канала, открываемого СВ pipe(). */
    /* Максимальная длина имени файла в UNIX-подобных ОС - 256 байт, включая
       завершающий нулевой символ, который показывает конец имени. Выделение памяти
       под символьные массивы для хранения имени файла name и буфера ввода-вывода
       buf: */
    char name[256], buf[80];

    /* Поскольку могут быть предоставлены грязные страницы памяти, то прежде, чем
       использовать выделенную память, желательно ее очистить с помощью функции
       memset() или bzero(), занося нулевые значения в область памяти: */
    memset(name, 0, 256);
    memset(buf, 0, 80);

    /* Настройка обработчика сигнала: */
    /* Выделение памяти под объект типа struct sigaction, который содержит 3 поля.
       В объекте prer будет указан новый метод обработки сигнала: */
    struct sigaction prer;

```

```

/* Задаются поля структуры struct sigaction:
   В поле sa_handler указывается имя (адрес) функции обработки сигнала,
   описанной в начале программы */
prer.sa_handler = hand1;
/* Поле sa_flags = 0, т.е. предусматривается стандартная обработка сигнала,
   который будет указан в СВ sigaction(). */
prer.sa_flags = 0;
/* Сохранение текущей СМ. Первые 2 аргумента нулевые, т.е. СМ текущего
   процесса не будет изменяться, а будет записана в переменную prer.sa_mask.
   Поскольку при создании процесс-потомок наследует СМ своего родителя, то
   весь перечень сигналов, которые поддерживаются ОС, отражен в СМ процесса,
   и ОС должна на него адекватно реагировать. */
sigprocmask(0, 0, &prer.sa_mask);
/* После того, как заполнены все три поля нового метода обработки сигнала
   prer, исполняется СВ sigaction(), первым аргументом которого является
   SIGINT (код 2) - сигнал прерывания < Ctrl+C > с терминала, по умолчанию
   вызывающий завершение процесса.

   Вторым аргументом является prer - новый метод обработки сигнала.
   Третий аргумент 0 означает, что старый метод обработки сигнала не будет
   сохранен, т.е. он не будет использоваться в дальнейшем.

   Поле prer.sa_mask определяет набор сигналов, которые будут добавлены к СМ
   процесса перед вызовом функции-обработчика prer.sa_handler = hand1(). Перед
   возвратом из обработчика сигнала СМ будет автоматически восстановлена в
   прежнее состояние. Таким способом можно блокировать определенные сигналы
   на время работы функции-обработчика.

   Перед доставкой сигнала, когда вызывается функция-обработчик, сам сигнал
   также включается в СМ; тем самым на время выполнения обработчика
   блокируется доставка того же самого сигнала. Такой подход гарантирует
   то, что во время обработки последующее поступление определенных сигналов
   будет приостановлено до завершения вызова. */
sigaction(SIGINT, &prer, 0);
/* СВ sigaction() описан в начале программы, поэтому, если в дальнейшем поступит
   сигнал SIGINT, то программа всегда на него отреагирует: любое нажатие
   < Ctrl+C > на клавиатуре вызовет функцию обработки сигнала hand1(), которая
   при каждом прерывании будет увеличивать счетчик сигналов, а при поступлении
   3-х сигналов прерывания выведет все файлы текущего каталога, написанные на
   языке программирования С или С++ и завершит выполнение программы. */

/* Тело основной программы */
/* Для повторения запроса на ввод от пользователя в случаях, когда предыдущие
   введенные пользователем значения не прошли проверку, а также когда получен
   сигнал во время ввода (пользователь ввел часть информации, но вдруг поступил
   сигнал), можно было бы применить библиотечные функции безусловного перехода
   sigsetjmp() и siglongjmp(). Однако в данном случае выбран бесконечный цикл,
   ждущий ввода имени существующего файла. Если во время ввода поступит сигнал,
   то бесконечный цикл обеспечит повторение запроса на ввод от пользователя.

   Также, пока не будет введено имя существующего файла, запрос на ввод будет
   повторяться снова и снова: */
while (1)
{

```

```

printf("Введите имя файла: ");
// Функция scanf() считывает с клавиатуры цепочку символов до пробела.
scanf("%s", name); // Ввод имени файла, не содержащего пробелы
/* Альтернатива 1:
   Если в именах файла предполагается использовать пробелы, то функцию
   scanf() можно заменить на СВ read(), который прочтет цепочку заданной
   длины из любых символов:
   Сброс всех буферов для того, чтобы запрос на ввод имени файла, выведенный
   через функцию printf(), успел появиться до чтения имени файла через
   СВ read(): */
// fflush(NULL);
// read(0, name, 256); // Ввод имени файла, содержащего пробелы
/* Однако, если пользователь введет слишком длинную цепочку символов,
   оказавшихся неверным именем файла, то в следующей итерации СВ read()
   продолжит считывать оставшиеся символы как новый ввод пользователя.
   Это делает использование этой функции неудобным. */
/* Альтернативы 2 и 3:
   Если в именах файла предполагается использовать пробелы, то удобнее
   применить функцию gets(), считающую строку из любых символов из
   стандартного ввода до символа перевода строки '\n' (или до конца файла)
   и записывающую ее в name: */
// gets(name); // Ввод имени файла, содержащего пробелы
/* Однако функция gets() не позволяет определить размер приемного буфера
   name, поэтому использовать ее не следует. Если входная строка окажется
   длиннее буфера, то это приведет к его переполнению и порче данных,
   которые находятся в памяти сразу после буфера.
   Поэтому в данном случае целесообразнее использовать функцию fgets(): */
// fgets(name, 256, stdin); // Ввод имени файла, содержащего пробелы

/* Посредством СВ access() проверяется, существует ли данный файл в текущей
   директории. Константа F_OK для СВ access() означает проверку на то,
   существует ли файл? Для того, чтобы проверить, разрешен ли к файлу доступ
   на чтение, в СВ access() нужно задать константу R_OK.
   Если введено корректное имя файла и этот файл существует, */
if (!access(name, F_OK))
    // то бесконечный цикл прерывается
    break;
/* ...В противном случае файл не существует, тогда бесконечный цикл
   продолжится, и запрос на ввод имени файла отправится снова.
   Если прерывание поступит во время ввода внутри этого цикла, то корректное
   имя файла не будет сформировано и снова произойдет возврат к запросу на
   ввод имени файла. */
} // Бесконечный цикл while(1)

/* После ввода имени существующего файла СВ pipe() создает межпроцессный канал и
   открывает два пользовательских дескриптора файлов - p[0] с доступом к каналу
   на чтение и p[1] с доступом на запись, которые будут записаны в ТПДОФ
   (таблицу пользовательских дескрипторов открытых файлов) процесса под новыми
   номерами 3 и 4, соответственно: */
pipe(p);

```

```

/* После создания межпроцессного канала процесс с помощью СВ fork() разделяется
на две идентичные копии, которые продолжают выполняться как два независимых
процесса:
1) Процесс-потомок получает от СВ fork() код ответа 0.
2) Процесс-родитель - идентификатор, под которым запущен процесс-потомок.
В случае аварийного завершения СВ fork() вернет -1. */

if (fork() == 0)
{ /* Ветка для потомка после успешного вызова fork():
   Из ТПДОФ процесса-потомка удаляется запись с номером p[0] = 3, т.е.
   закрывается доступ к межпроцессному каналу на чтение: */
    close(p[0]);
   /* Теперь потомок может только передавать информацию родителю...
      Закрытие дескриптора файла стандартного вывода STDOUT_FILENO = 1
      (из ТПДОФ процесса-потомка удаляется запись с номером 1): */
    close(1);
   /* Отождествление стандартного вывода с файловым дескриптором канала,
      предназначенным для записи. Для этого посредством СВ dup() выполняется
      копирование дескриптора канала, предназначенного для записи (запись с
      номером p[1] = 4), в первую (наименьшую) свободную запись в ТПДОФ
      процесса (начиная от 0), т.е. в STDOUT_FILENO = 1, поскольку
      STDIN_FILENO = 0 занят, а дескриптор файла стандартного вывода с номером
      1 был предварительно закрыт. */
    dup(p[1]);
   /* Осуществлено перенаправление вывода. Вместо файла стандартного вывода,
      которым является монитор, установлен межпроцессный канал, который открыт
      на запись, поэтому результат выполнения любой стандартной инструкции,
      функции или утилиты будет выводиться не на экран монитора, а в
      межпроцессный канал.

Выполнение утилиты ls с ключом -la, которая выводит список с подробной
информацией обо всех файлах (владелец, группа, права доступа, время
последнего обновления, размер и др.) текущего каталога, включая те файлы,
имена которых начинаются с точки (скрытые). Результат выполнения этой
инструкции вместо монитора будет выведен в межпроцессный канал.

Утилита ls выполняется с помощью СВ exec1(), аргументы которому задаются
в виде списка переменной длины, который должен заканчиваться значением
NULL, чтобы СВ exec1() смог определить, какой именно аргумент является
последним. Первый аргумент СВ exec1() - это полное имя программы ls,
которая находится в папке bin, затем идет нулевой внешний аргумент
(собственно, само имя программы ls), затем идут флаги -la и -d команды
ls, затем в качестве второго внешнего аргумента используется имя файла,
которое ввел пользователь. */
    exec1("/bin/ls", "ls", "-la", "-d", name, NULL);
   /* СВ exec1() убивает процесс, в рамках которого он был запущен, т.е.
      вызывает новую программу вместо уже выполняющейся без возврата в
      вызывающую программу. Выполнив новый процесс "ls -la -d ИМЯФАЙЛА", он
      запишет результат его выполнения в межпроцессный канал вместо монитора.
   */
}
else

```

```

{ /* Ветка для родителя после успешного вызова fork(): */
/* Родитель приостанавливает свою работу и ждет завершения процесса-потомка,
   вызвав СВ wait(). Аргумент 0 означает, что причина гибели процесса-
   потомка (например, по СВ exit() или по сигналу) нам не интересна).
   Поскольку процесс-потомок был всего один, то возвращаемое значение
   СВ wait() также неважно: */
wait(0);
/* После завершения процесса-потомка, из ТПДОФ процесса-родителя удаляется
   запись с номером p[1] = 4, т.е. закрывается доступ к межпроцессному
   каналу на запись: */
close(p[1]);
/* Теперь родитель может только читать информацию от потомка...
   Бесконечный цикл, обрабатывающий результаты работы процесса-потомка: */
while (1)
{ /* Считывание данных из межпроцессного канала в массив buf по 80 байтов.
   Если прочитан хотя бы один байт, то */
if (read(p[0], buf, 80) != 0)
{ /* считанные данные обрабатываются - определяется тип файла
   по первому символу (первому байту) в массиве buf: */
if (buf[0] == '-')
    printf("файл - обычный\n");
if (buf[0] == 'd')
    printf("файл директория\n");
if (buf[0] == 'r')
    printf("файл канал\n");
if (buf[0] == 'c')
    printf("файл специальный байт-ориентированный\n");
if (buf[0] == 'b')
    printf("файл специальный блок-ориентированный\n");
if (buf[0] == 's')
    printf("файл гнездо\n");
if (buf[0] == 'l')
    printf("файл символьная ссылка\n");
/* Очистка символьного массива buf с помощью функции memset() путем
   занесения нулевых значений в область памяти: */
memset(buf, 0, 80);
}
/* Когда СВ read() вернет 0, будет достигнут конец файла межпроцессного
   канала p[0]: */
else
    break;
/* Завершается выполнение основной программы. Библиотечная функция
   exit() сбрасывает буферы ввода-вывода и завершает процесс, закрывая
   все открытые в процессе файлы; код 0 означает правильное завершение
   процесса-родителя: */
exit(0);
} // Бесконечный цикл while(1)
}
}

```

Пример реализации. Итог

```

1 #include <stdio.h>
2 #include <signal.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5 #include <setjmp.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <sys/wait.h>
9 sigjmp_buf abuf;
10 int count=0;
11 void handle() {
12     int p[2], p1[2];
13     count++; // Подсчет количества принятых сигналов
14     if( count==3 ) {
15         pipe(p);
16         if(fork() == 0) {
17             close(p[0]);
18             dup2(p[1], 1);
19             system("file *");
20             _exit(0);
21         } else {
22             wait(0);
23             pipe(p1);
24             close(p[1]);
25             if(fork() == 0) {
26                 close(p1[0]);
27                 dup2(p1[1],1);
28                 dup2(p[0], 0);
29                 system("grep \"C source\"");
30                 _exit(1);
31             } else {
32                 wait(0);
33                 close(p1[1]);
34                 dup2(p1[0], 0);
35                 write(1, "\n", 1);
36                 system("cut -d : -f 1");
37                 _exit(0);
38             } } }
39 void main()
40 {int fd; int p[2]; char name[256], buf[80];
41 struct sigaction prer;
42 memset( name, 0, 256 ); memset( buf, 0, 80 );
43 prer.sa_handler=handle;
44 prer.sa_flags=0;
45 sigprocmask( 0, 0, &prer.sa_mask );
46 sigaction( SIGINT, &prer, 0 );
47 while(1)
48 {
49     printf( "Введите имя файла:" );
50     scanf( "%s", name );
51     if(!access(name, F_OK)) break;
52 }
53 pipe(p);
54 if( fork() == 0 )
55 {
56     close(p[0]);
57     close( 1 );
58     dup( p[1] );
59     execl( "/bin/ls", "ls", "-la", "-d", name, NULL );
60 } else {
61     wait(0);
62     close(p[1]);
63     while(1{
64         if(read(p[0], buf, 80) != 0)
65             {if( buf[0] == '-' ) printf( "файл - обычный\n");
66              if( buf[0] == 'd' ) printf( "файл директория\n");
67              if( buf[0] == 'p' ) printf( "файл канал\n");
68              if( buf[0] == 'c' ) printf( "файл специальный байт-ориентированный\n");
69              if( buf[0] == 'b' ) printf( "файл специальный блок-ориентированный\n");
70              if( buf[0] == 's' ) printf( "файл гнездо\n");
71              if( buf[0] == 'l' ) printf( "файл символьная ссылка\n");
72              memset( buf, 0, 80 ); } else break; exit( 0 );
73 } } }
```

Примечание: на скриншоте присутствуют два лишних заголовочных файла и лишняя переменная abuf:

```
#include <sys/types.h>
#include <setjmp.h>
/* Нелокальные переходы. Содержит
прототипы функций безусловного
нелокального перехода sigsetjmp()
и siglongjmp(), а также определения
связанных с ними структур данных */
sigjmp_buf abuf;
```

Кроме того, здесь, также как в «11.2 Пример реализации», присутствует переменная fd, которая в программе затем никак не используется.

12. Распределение и перераспределение оперативной памяти в ОС UNIX.

Программные средства и системные вызовы работы с оперативной памятью

Распределение и перераспределение оперативной памяти — последняя функция ядра, которую мы будем рассматривать. ОП — неотъемлемая часть любой вычислительной системы, т.к. все действия, которые реализуются в компьютере, выполняются с использованием ОП. Между одновременно работающими процессами память надо делить — после кванта времени она отнимается у одного процесса и дается другому процессу. ОС следит за тем, какая память свободна и какую память надо забрать у процесса.

Существует несколько подходов к распределению ОП:

- На основе свопинга.
- На основе страничной подкачки.

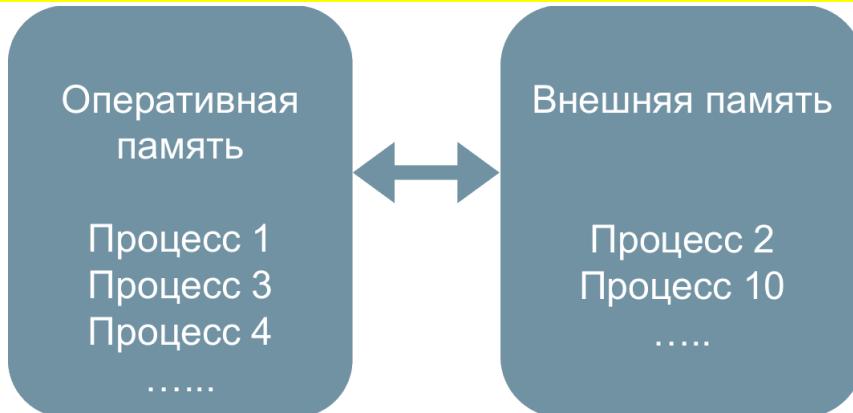
12.1 Распределение оперативной памяти на основе свопинга

Большинство систем UNIX для управления памятью реализуют подход свопинга (выгрузке из ОП во внешнюю память). Этот подход сразу был заложен в системах UNIX начиная с 1960 года. Свопинг не требует, чтобы программа оставалась в оперативно памяти до момента завершения. Таким образом, если ОП одновременно не вмещает все параллельно работающие процессы, то часть этих процессов можно располагать во внешней памяти. Поскольку ОС должна выполнять каждый из присутствующих процессов, то между ОП и внешней памятью постоянно должен производиться обмен. Все процессы для ОС равноправны (все процессы считаются активными) и она выполняет их в соответствии с приоритетом каждого указанного процесса. Не может быть такого, что часть процессов ОС будет выполнять, а часть держать в ОП в качестве «изгоев».

Свопинг / swapping

Перемещением данных между памятью и диском (свопингом) управляет верхний уровень планировщика, называемый swapper (процесс с идентификатором PID = 0). Задача этого процесса — выгрузка процессов из ОП во внешнюю память и наоборот, загрузка процессов из внешней памяти в оперативную.

Swap — вторичная память или область свопинга. Область, как правило, дискового пространства, где могут сохраняться временно не используемые участки адресного пространства процесса.



Свопинг. Выгрузка данных во внешнюю память

Выгрузка данных из ОП во внешнюю память инициируется, когда у ядра кончается свободная память из-за одного из следующих событий:

1. СВ `fork()` требуется память для процесса-сына.
2. СВ `brk()` собирается расширить сегмент данных какого-то процесса, а памяти под расширенное адресное пространство процесса может не хватить.
3. Динамическому сегменту требуется дополнительная память (`malloc()`, `calloc()`) и т.п. — т.е. выделение дополнительной памяти под выделенный указатель). Размер процесса увеличился в результате естественного увеличения стека процесса.

4. Продолжение выполнения процесса, находящегося во внешней памяти. Ядру нужно освободить в памяти место для подкачки ранее выгруженных процессов. Когда наступает время запустить процесс, долго находящийся на диске, часто бывает необходимо удалить из памяти другой процесс, чтобы освободить место для запускаемого процесса.

Подробнее см. [здесь](#) и Робачевский А., Немюгин С., Стесик О. Операционная система UNIX. 2-е изд. — СПб.: БХВ-Петербург, 2010 г., стр. 252-254.

Свопинг. Выгрузка данных из оперативной памяти

Если нужно что-то выгрузить из ОП, то выгружаются процессы с наивысшим значением суммы приоритета и времени пребывания в ОП:

1. Выгружается процесс в состоянии «блокирован».

Выбирая «жертву», т.е. процесс (или группу процессов), который надо выгрузить из ОП, **swapper рассматривает блокированные процессы**. Эти процессы, **ожидающие наступления некоторого события** (например, ввода-вывода информации, завершения другого процесса, закрытия какого-либо файла, срабатывания таймера), в данный момент **не работают и не будут работать, пока не выйдут из состояния блокировки**. Если они нашлись, то из них выбирается процесс с наивысшим значением суммы приоритета и времени пребывания в ОП.

2. Выгружается процесс в состоянии «готов».

Если блокированных процессов нет, то на основе этого же критерия выбирается готовый процесс. **Готовый процесс** — процесс, который может сразу использовать предоставленный ему ЦП, но в данный момент ему пока не выделен ЦП.

Почему же в качестве критерия выбирается значение суммы приоритета и времени пребывания в ОП? Напомним, что **приоритет — целое число. Чем выше это число, тем ниже приоритет у процесса. Поэтому чем выше сумма приоритета и времени пребывания в ОП, тем меньше этот процесс востребован и его можно выгружать из памяти.**

Свопинг. Загрузка процессов из внешней памяти в оперативную

1. Периодически (каждые несколько секунд) **swapper** исследует список выгруженных во внешнюю память процессов, проверяя, не перешел ли какой-нибудь из них из состояния «блокирован» в «готов».
2. Если процессы в состоянии готовности обнаружены, то из них выбирается процесс с состоянием «готов», дольше всех (по времени) находящийся во внешней памяти (на диске).
3. Затем **swapper** проверяет, какая процедура свопинга будет применена:
 - **Лёгкий свопинг** не требует дополнительного высвобождения ОП для вновь появившегося процесса. Т.е. ОС перекачивает процесс из внешней памяти в ОП и свободного размера ОП достаточно для осуществления этой операции.
 - **Тяжелый свопинг** — это свопинг, при котором для загрузки в ОП выгруженного на диск процесса, из неё требуется выгрузить один или несколько других процессов.
4. Выбранный во внешней памяти процесс загружается в ОП.

Свопинг. Загрузка процессов из внешней памяти в оперативную. Периодичность

Описанную выше последовательность действий по загрузке процессов из внешней памяти в ОП постоянно осуществляет **swapper** (процесс с идентификатором PID = 0). Загрузка процессов из внешней памяти в ОП прекращается (приостанавливается) при выполнении одного из условий:

1. Во внешней памяти (на диске) нет процессов, готовых к работе (в состоянии «готов»).
2. В ОП не осталось места для новых процессов.

Чтобы не терять большую часть производительности системы на свопинг, **ни один процесс не выгружается на диск, если он находится в памяти менее 2-х секунд**. Это время часто меняют системные администраторы, но в целом оно подобрано оптимально.

12.2 Распределение оперативной памяти на основе страничной подкачки

Чистый свопинг в современных ОС уже практически не используется, а используется комбинированный метод на основе свопинга со страничной подкачкой. Практически во всех системах UNIX к системе управления памятью добавлена страничная подкачка. Это нужно для предоставления возможности работы с программами больших размеров.

Форматирование оперативной памяти

Вся оперативная память разбивается на блоки, т.к. побайтовый учет ОП требует больших накладных расходов (в ОП приходится содержать очень большие таблицы).

Если блоки имеют одинаковый размер — они называются страницами. Типовой размер страницы памяти во многих современных ОС — 4 КБ.

Если блоки имеют разный размер — они называются сегментами.

Учет свободного места в оперативной памяти и во внешней памяти. Битовая карта

Способы учета распределения памяти, разбитой на блоки:

1. Битовая карта

Битовая карта представляет собой состоящую из двух полей (номер блока и бит) таблицу, в которой для каждого блока указан бит. Если бит в состоянии 1, то блок занят, если бит в состоянии 0, то блок свободен:

| Номера блоков | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|----------------|---|---|---|---|---|---|---|---|------|
| Значения битов | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | |

Учет свободного места в оперативной памяти и во внешней памяти. Связный список свободных/занятых блоков

2. Связный список свободных (иногда занятых) блоков

Список состоит из записей. Каждая запись в связном списке содержит адрес, с которого начинается область свободных блоков, длину области (число свободных блоков), указатель на следующую запись (на следующую свободную область памяти):

| | | | | | | | | | |
|----------|-----------|-----------|-----|-----|-----|-----|-----|-----|-----|
| 1, 2, 10 | 10, 1, 14 | 14, 5, 22 | ... | ... | ... | ... | ... | ... | ... |
| 1 запись | 2 запись | 3 запись | ... | ... | ... | ... | ... | ... | ... |

С помощью такого списка удобно учитывать распределение памяти. Связанные список свободных блоков обычно используется при распределении внешней памяти, поскольку обычно ее объем достаточно большой и, если ее отображать с помощью битовой карты, то потребуется достаточно большой объем самой битовой карты. Поскольку битовая карта должна располагаться в ОП, то в этом случае она занимала бы лишний объем ОП.

Итак, свободное место в памяти и на диске учитывается при помощи связанного списка. Когда требуется свободное пространство в памяти или на диске (во внешней памяти), из списка выбирается первое подходящее. После этого, в список возвращается остаток от свободного пространства.

Свопинг со страничной подкачкой

Суть этого подхода заключается в следующем: для того, чтобы работать, процессу не нужно целиком находиться в ОП. Процессу для работы требуется держать в ОП только контекст процесса и таблицы страниц:

1. Контекст процесса.

2. Таблица страниц процедурного сегмента.
3. Таблица страниц сегмента данных.
4. Таблица страниц динамического сегмента.

Если они загружены, то процесс считается находящимся в памяти и может быть запущен планировщик. Сами страницы с процедурным сегментом, сегментом данных и динамическим сегментом подгружаются в ОП динамически во время выполнения процесса, по мере обращения к ним. Т.е. когда процесс начинает выполняться и какая-то из его страниц не находится в ОП, происходит страничное прерывание. По этому прерыванию в случае нехватки ОП выбирается блок из связанного списка и в этот блок перекачивается необходимая страница данного сегмента.

Страницная подкачка

Страницная подкачка реализуется двумя процессами:

- **Диспетчерским процессом** с PID = 0, который отвечает как за планирование процессов, так и за свопинг (*swapper*), т.е. он выгружает процессы из ОП во внешнюю память и наоборот. Он уже был рассмотрен ранее в «2.6 Планирование процессов в ОС UNIX».
- Процессом с PID = 2, называемым **страницным демоном (page demon)**. Он отвечает за список свободных страниц (за выгрузку страниц из ОП во внешнюю память). Реализует алгоритм замещения страниц. *Страницный демон периодически запускается и смотрит, есть ли для него работа. Если он обнаруживает, что количество страниц в списке свободных страниц мало, то страницный демон инициирует действие по освобождению дополнительных страниц, перенося страницы из ОП на диск.*

Распределение оперативной памяти

Оперативная память в ОС семейства UNIX целиком состоит из блоков или страниц памяти и условно **делится на 3 части**:

- **Ядро ОС** постоянно находится в ОП. Ядро ОС — набор программ и структур данных (таблиц), которые поддерживают состояние всей вычислительной системы.
- **Карта (таблица) памяти**, с помощью которой ведется учет всей ОП — также постоянно находится в ОП. Т.е. в этой таблице написано, какая страница памяти и чем она занята — это может быть **свободная страница памяти, или это может быть страница памяти, относящаяся к одному из выполняемых в данное время процессов.**
- **Страницы памяти** — могут быть свободными, или занятыми каким-то из процессов, который выполняется в данный момент времени. Т.е. страницы памяти могут либо быть свободными, либо подвергаться выгрузке во внешнюю память, либо подвергаться принятию информации из внешней памяти.

Первые две — **ядро ОС и карта памяти** — резидентны, т.е. **никогда не выгружаются из ОП**. Остальная память делится на страницы, каждая из которых может содержать страницы процедурного сегмента, сегмента данных и динамического сегмента того или иного процесса, или находиться в списке свободных страниц.

Карта оперативной памяти

Карта оперативной памяти UNIX-подобной ОС содержит информацию о содержимом страниц. По сути это таблица, у которой девять полей. Для каждой страницы в карте памяти есть запись, состоящая из девяти полей:

- Первые два поля записи карты памяти используются только тогда, когда соответствующая страница находится в списке свободных страниц. В этом случае они «сшивают» свободные страницы в двухсвязный список свободных страниц. Если в карте памяти первые два поля заняты (как на рисунке выше), то значит эта страница относится к числу свободных страниц ОП. Если же первые два поля свободны, то эта страница памяти считается занятой и ее характеризуют последние поля.

- Следующие три блока используются, когда страница содержит информацию. У каждой страницы в оперативной памяти есть фиксированное место хранения на диске (во внешней памяти), в которое она помещается, когда выгружается из ОП.
- Следующие три поля содержат ссылку на запись в таблице процессов, тип хранящегося в таблице сегмента (процедурный, сегмент данных или динамический) и смещение в данном сегменте, т.е. номер страницы в сегменте.
- Последнее поле содержит флаг (бит использования), необходимый для алгоритма замещения страницы (алгоритм часов), выталкивающий страницы из ОП во внешнюю память.

| Номер следующей страницы | Номер предыдущей страницы | Номер блока диска | Номер диска | Хэш-код блока | Номер дескриптора процесса | Тип сегмента | Смещение в сегменте | Бит использования |
|--------------------------|---------------------------|-------------------|-------------|---------------|----------------------------|--------------|---------------------|-------------------|
| 10 | 1 | | | | | | | |

| | | | | | | | | |
|------------------------|-------------------------|-------------------|------------------------|---------------|------------------------|-----------------------|---------------------|--------------------|
| Номер следующей записи | Номер предыдущей записи | Номер блока диска | Номер блока устройства | Хэш-код блока | Идентификатор процесса | Тип сегмента процесса | Смещение в сегменте | Флаг использования |
|------------------------|-------------------------|-------------------|------------------------|---------------|------------------------|-----------------------|---------------------|--------------------|

Подкачка страниц в оперативную память

Процесс никогда целиком не находится в ОП, находится только часть процесса. Процесс во время своей работы может содержать не одну и не две, а целый десяток страниц ОП. Выгрузка процесса на диск может произойти и целиком, в зависимости от того, сколько страниц он занимает в ОП.

При выполнении процесс может вызвать страничное прерывание, если одной или нескольких его страниц не окажется в ОП.

При страничном прерывании ОС берет 1-ю страницу из списка свободных страниц ОП, удаляет ее из списка и подгружает в нее требуемую страницу из внешней памяти.

Если список свободных страниц пуст, то выполнение процесса приостанавливается до тех пор, пока страничный демон не освободит необходимое количество страниц.

12.3 Замещение страниц оперативной памяти

Замещением страниц ОП (выгрузкой страниц из ОП во внешнюю память) занимается **страничный демон** (процесс с идентификатором PID = 2, например, /proc/sys/vm/swappiness). Каждые 250 мс он просыпается для того, чтобы сравнить количество свободных страниц ОП с системным параметром **lostfree** ($\sim \frac{1}{4}$ объема ОП в Linux). В некоторых UNIX-подобных ОС (например, в BSD — Mac OS, Solaris) используются 2 параметра — **min** ($\sim \frac{1}{4}$) и **max** ($\sim \frac{1}{2}$).

Если количество свободных страниц $<$ **lostfree** (или **min**), страничный демон начинает переносить страницы из ОП во внешнюю память (на диск) до тех пор, пока количество свободных страниц не станет равным **lostfree** (или **max**).

Алгоритм замещения страниц

Идеальный алгоритм замещения страниц заключается в том, чтобы **выгружать ту страницу, которая будет запрошена позже всех**. Но этот алгоритм **не осуществим**, т.к. нельзя знать какую страницу, когда запросят. Можно лишь набрать статистику использования.

Чтобы избежать перемещения страниц по списку, можно использовать указатель, который перемещается по списку, что значительно быстрее модификации всего списка. Для этого записи страниц хранят в кольцевом списке и используют указатель на одну из ячеек. Кольцевой список используется в т.н. основном алгоритме часов, в котором все занятые страницы укладываются на циферблат. Итак, **основной алгоритм часов работает, сканируя занятые страницы, как если бы они лежали на окружности циферблата.**

1. На первом проходе, когда стрелка часов указывает на страницу карты памяти, сбрасывается её флаг использования.
2. На втором проходе у каждой страницы, к которой не обращались с момента первого прохода, флаг использования остаётся сброшенным и эта страница будет помещена в список свободных страниц (выгружена во внешнюю память).

Страница в списке свободных страниц сохраняет своё содержание, что позволяет восстановить её, если она потребуется, прежде чем она будет перезаписана.

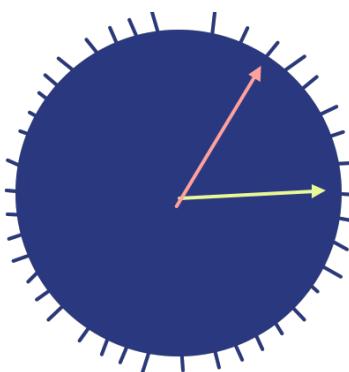
В ОС семейства Windows применяется алгоритм **NRU** (Not Recently Used — не использовавшаяся в последнее время страница). В таблице страниц ОП для каждой страницы присутствуют 2 бита:

- бит R (бит обращения, Referenced) выставляется в 1 при каждом обращении к странице. Через некоторое время (например, каждые n тиков таймера) ОС переводит его в 0, чтобы отличить страницы, к которым давно не было обращения;
- бит M (бит модификации, Modified) выставляется в 1 при изменении страницы. Сигнализирует о том, что при удалении надо страницу записать на диск. M переводится в 0 только после записи на диск.

При страничном прерывании, на основании значений битов R и M, ОС делит все страницы на 4 класса:

1. не было обращений и изменений ($R=0, M=0$);
2. не было обращений, было изменение ($R=0, M=1$);
3. было обращение, не было изменений ($R=1, M=0$);
4. было обращений и изменений ($R=1, M=1$).

Для удаления случайным образом выбирается страница из низшего класса. Алгоритм легок в реализации и может дать вполне достаточный результат.



Страницный демон в ОС семейства UNIX использует **модифицированный алгоритм часов**, в котором есть сразу 2 «стрелки» (указателя). В этом алгоритме страницный демон поддерживает 2 указателя на карту памяти:

- 1-ый указатель сбрасывает бит использования данной страницы в карте памяти.
- 2-ой указатель (обычно идет через несколько (100-150) блоков) проверяет, не использовалась ли эта страница в течении заданного интервала времени. Если она не использовалась, то ее можно выгрузить из ОП во внешнюю память.

При работе алгоритм сначала сбрасывает флаг использования передней стрелкой, а затем проверяет этот флаг задней стрелкой. При каждом запуске страницного демона стрелки проходят не полный оборот, а столько, сколько необходимо, чтобы количество страниц в списке свободных страниц было не меньше **lostfree** (или **max**).

Модифицированный алгоритм часов очень похож на алгоритм NRU (Not Recently Used) тем, что из ОП выгружаются во внешнюю память те страницы, которые не использовались в последнее время, однако в алгоритме часов страницы выгружаются по блочному принципу, а в NRU по временному параметру.

Поддержка свопинга

Если частота подкачки страниц слишком высокая (алгоритм замещения постоянно освобождает страницы), а количество свободных страниц ОП ниже **lostfree** (или **min**), то ОС принимается решение о том, что из ОП надо выгружать уже не отдельные страницы, а целые процессы. Тогда swapper (процесс с PID = 0) начинает выгружать из ОП один или несколько процессов.

В этом случае ОС должна выбрать, какие именно процессы выгрузить из ОП для того, чтобы обеспечить необходимый объем свободных страниц ОП.

Критерии выбора процесса на удаление из оперативной памяти

Swapper руководствуется следующими критериями:

0. Как известно из «12.1 Распределение оперативной памяти на основе свопинга», из ОП выгружаются процессы с наивысшим значением суммы приоритета и времени пребывания в ОП. В первую очередь выгружается те процессы, которые находятся в состоянии «блокирован». Если блокированных процессов нет, то выбираются процессы в состоянии «готов»:
 1. Удаляется процесс, который не проявлял активности (бездействовал) в течении 20 и более секунд. Если такие процессы есть, то из них выбирается с максимальным временем бездействия и выгружается на диск.
 2. Если таких процессов нет, то анализируются 4 самых больших процесса (по количеству страниц в ОП) и выгружается тот, который находится в памяти дольше всех.

Процессы удаляются из памяти до тех пор, пока не освободится достаточно большое количество памяти.

Каждые несколько секунд swapper проверяет, есть ли на диске готовые процессы, и выбирает процесс с наивысшим приоритетом. Загрузка выбранного процесса производится только при наличии достаточного количества страниц, чтобы, когда случится неизбежное страничное прерывание, для него нашлись бы свободны страничные блоки. Swapper загружает в ОП только контекст процесса и таблицы страниц. Сами страницы с процедурным сегментом, сегментом данных и динамическим сегментом подгружаются в ОП при помощи обычной страничной подкачки динамически во время выполнения процесса, по мере обращения к ним.

Область свопинга

У каждого сегмента каждого процесса есть место во внешней памяти, где он располагается, когда его страницы выгружаются (удаляются) из оперативной памяти.

Сегмент данных и динамический сегмент каждого процесса сохраняются в виде временной копии в области свопинга. Причем часто динамический сегмент не подвергается свопингу.

Процедурный сегмент подгружается из самого исполняемого двоичного файла во внешней памяти.

12.4 Системные средства манипулирования оперативной памятью. Системные вызовы

Страницы ОП, которые выделяются под новые процессы, не всегда могут оказаться пустыми, т.е. они могут содержать какую-то информацию. На жаргоне их называют **грязными страницами памяти**. В ОС существуют процессы или потоки (если ОС поддерживает потоки), которые несут ответственность за очистку перераспределенных страниц ОП. В рамках UNIX-подобных ОС обычно этим занимаются от 2 до 8 потоков. Однако не всегда они успевают очистить память. Поэтому, **после того, как вы в своих программах получаете память под переменные или массивы, то прежде чем использовать выделенную память, не забывайте ее очистить с помощью функции `memset()` или `bzero()`**, занся в область памяти нулевые значения.

Рассмотрим несколько СВ, которые можно использовать для перераспределения ОП в рамках своих программных приложений. СВ `fork()` уже был рассмотрен ранее.

Системный вызов brk()

Изменение размеров кучи (то есть выделение и высвобождение ОП) сводится лишь к тому, чтобы объяснить ядру, где располагается крайняя точка программы (program break).

СВ **brk()** и **sbrk()** — это базовые низкоуровневые вызовы UNIX для динамического выделения памяти. Они изменяют размер сегмента данных процесса или, если быть более точным, смещают верхнюю границу сегмента данных процесса. Увеличивая или уменьшая сегмент данных процесса вы увеличиваете или уменьшаете адресное пространство процесса. Вызов **brk()** устанавливает абсолютное значение границы сегмента, а вызов **sbrk()** — ее относительное смещение.

В большинстве ситуаций для выделения динамической памяти рекомендуется использовать функции семейства **malloc**, а не эти вызовы.

СВ **brk()** устанавливает крайнюю точку процесса на место, указанное окончанием сегмента данных — **end_data_segment**.

```
#include <unistd.h>
int brk(void *end_data_segment)
*end_data_segment — указатель, представляющий окончание сегмента данных процесса. Поскольку виртуальная память выделяется постранично, end_data_segment фактически округляется до границы следующей страницы.
```

Возвращает **0** при успешном завершении или **-1** при ошибке.

Функции **malloc()**, **calloc()**, **realloc()** и **free()** при использовании СВ **brk()** лучше не применять!

Вызов (или функция) **sbrk()** приводит к изменению положения точки программы путем добавления к ней приращения **increment** байт:

```
#include <unistd.h>
void *sbrk(intptr_t increment);
```

Используемый для описания приращения **increment** тип **intptr_t** является целочисленным типом данных.

Возвращает предыдущий адрес крайней точки программы при успешном завершении или (**void ***) **-1** при ошибке.

Иными словами, если мы подняли крайнюю точку программы, то **возвращаемым значением будет указатель на начало только что выделенного блока памяти**.

Вызов **sbrk(0)** возвращает текущее значение установки крайней точки программы без ее изменения. Этот вызов может пригодиться, если нужно отследить размер кучи, возможно, чтобы изучить поведение пакета средств выделения памяти.

Системный вызов mmap()

Для того, чтобы значительно ускорить процесс чтения информации из файла и записи информации в файл, СВ **mmap()** отображает указанные данные открытого файла в адресном пространстве вызывающего процесса. Один из его аргументов — пользовательский дескриптор открытого файла (из внешней памяти).

```
#include <unistd.h>
#include <sys/mman.h>
void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
Отображает указанную часть открытого файла (из внешней памяти) с номером пользовательского дескриптора fd в оперативную память. Файл должен быть заранее открыт при помощи СВ open(). Полученный дескриптор файла используется в качестве параметра fd. Отображает length байтов, начиная с указателя чтения/записи offset файла, по указателю стартового адреса start. Обычно в аргументе start передается 0, что позволяет ОС самой выбрать начальный адрес.
```

Аргумент **prot** описывает желаемый режим защиты памяти (**PROT_EXEC**, **PROT_READ**, **PROT_WRITE**, **PROT_NONE**). Для области памяти нельзя использовать степень защищенности, которая дает больше

прав доступа, чем позволяет режим, в котором открыт файл. Например, нельзя указать значение **PROT_WRITE**, если файл открыт только для чтения.

Аргумент **flags** задает тип отражаемого файла и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие.

Возвращает адрес начала области отображаемой памяти в случае успеха (никогда не бывает равным 0), **MAP_FAILED** — в случае ошибки

Отображения делятся на две категории.

- **Файловое отображение**, которое отображает область открытого файла на виртуальную память вызывающего процесса. После отображения содержимое файла может быть доступно с помощью операций над байтами в соответствующей области памяти. **Страницы отображения автоматически загружаются из файла по мере необходимости**. Отображенный файл (*mapped file*) — сегменты отраженного файла используются для того, чтобы отобразить части файлов в адресное пространство процесса, и использовать стандартные механизмы ОС управления виртуальной памятью для ускорения доступа к файлам.
- В противоположность первой категории, **анонимное отображение не имеет соответствующего файла**. Вместо этого страницы отображения получают начальное значение 0.

СВ **munmap()** применяется для того, чтобы выгрузить отображенный с помощью СВ **mmap()** файл из ОП:

```
#include <unistd.h>
#include <sys/mman.h>
int munmap(void *address, size_t length);
```

Возращает **0** в случае успеха, **-1** — в случае ошибки. Если был задан флаг **MAP_SHARED**, то в файл вносятся все оставшиеся изменения, при флаге **MAP_PRIVATE** все изменения отбрасываются.

Следует иметь в виду, что **эта функция только отменяет отображение файла в память, но не закрывает файл**. Файл требуется закрыть при помощи СВ **close()**.

Системный вызов **shmget()**

Для создания объекта разделяемой памяти служит СВ **shmget()**. Эта разделяемая область памяти создается в области ядра ОС и к ней могут получить доступ все процессы, которые существуют в данный момент. Каждый из существующих процессов может присоединить этот участок ОП к своему адресному пространству и считывать или записывать в него информацию. Нужно только не допускать ситуаций, когда два и более процессов одновременно пишут или одновременно читают из этого участка ОП. Для этого при работе с участком разделяемой памяти необходимо синхронизировать доступ к нему. Т.е. если один процесс присоединил этот участок разделяемой памяти к своему адресному пространству и работает с ним (читает или пишет в него), то другие процессы должны подождать, пока этот ресурс не освободит процесс, захвативший его.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

Создает или открывает разделяемую область памяти с требуемым минимальным размером **size** (байт). Если необходимо создать новый участок разделяемой памяти (обычно на стороне сервера), его размер следует определить в аргументе **size**. Если нужно лишь получить ссылку на существующий участок (в случае клиента), в аргументе **size** можно передать **0**. Когда создается новый участок, его содержимое очищается.

Аргумент **key** — идентификатор участка памяти в системе (ключ для доступа к разделяемой памяти).

Параметр **shmflg** задает права доступа к участку памяти и определяет флаги, управляющие поведением вызова.

Возвращает **номер пользовательского дескриптора участка разделяемой памяти** (сегмент памяти в ОС семейства UNIX представляется также, как и открытый файл) в случае успеха, или **-1** в случае неудачи.

Системные вызовы **shmat()**, **shmdt()** и **shmctl()** для работы с разделяемой памятью

После создания участка разделяемой памяти (в результате вызова **shmget()**) процесс может присоединить его к своему адресному пространству с помощью СВ **shmat()**.

По окончании работы с участком разделяемой памяти следует вызывать СВ **shmdt()** для его отсоединения от адресного пространства процесса (это означает, что процесс больше не может использовать его).

*Обратите внимание: эта функция не удаляет из системы идентификатор и структуры данных, ассоциированные с участком памяти. Идентификатор продолжает существовать, пока какой-либо процесс (зачастую сервер) специально не удалит его вызовом **shmctl()** с командой **IPC_RMID**.*

СВ **shmctl()** выполняет различные операции над управляющими параметрами участка разделяемой памяти (размер памяти, владелец памяти, права доступа к этому участку памяти и т.д.).

В рамках ОС может быть создано большое число участков разделяемой памяти. ОС ведет таблицу, в которой отражены все эти участки разделяемой памяти.

12.5 Системные средства манипулирования оперативной памятью. Утилиты

Системные вызовы (СВ) применяются в программных кодах, когда вы разрабатываете программное приложение, а утилиты — это уже готовые программы, которые работают в командной строке и с помощью которых вы можете получить информацию о файле.

Как уже говорилось выше в «5.5 Системные вызовы и утилиты, позволяющие получить информацию о файле», в отличии от системных вызовов, **утилиты не имеют стандартов**. Каждую утилиту каждый разработчик не сумел стандартизировать. Поэтому в разных системах одни и те же утилиты могут работать **немного по-разному**. Поэтому прежде чем использовать утилиту в своих программных приложениях, надо исследовать, какие возможности предоставляет каждая отдельная утилита от каждого отдельного разработчика. Утилиты так же называют **командами**, а по своей сути это **готовые программные решения**. Утилиты, идущие в рамках ОС — это программы, которые помогают пользователю смотреть за различными действиями, которые осуществляют ОС.

Утилита free

Представляет данные о распределении всей оперативной памяти: сколько ОП всего, сколько занято, сколько отводится под буфер/кэш, сколько ОП доступно.

```
ubuntu@ubuntu:~/dim/one$ free
              total        used        free      shared  buff/cache   available
Mem:       873272       145636       429132          3976      298504       703000
Swap:          0          0          0
```

Утилита memstat

Отображает объемы оперативной памяти каждого активного процесса ОС. Отображает идентификаторы процессов и сколько памяти занимает данный процесс в настоящее время. Зная эту информацию, можно спрогнозировать, какой процесс ОС может подвергнуть свопингу, т.е. переместить из ОП во внешнюю память.

```
ubuntu@ubuntu:~/dim/one$ memstat
 1696k: PID 1769 (/usr/lib/systemd/systemd)
 1220k: PID 1880 (/usr/bin/bash)
 3968k: PID 2880 (/usr/bin/atop)
 296k: PID 2268 (/usr/bin/memstat)
 764k( 732k): /usr/bin/bash 1880
 580k( 512k): /usr/lib/arm-linux-gnueabihf/libzstd.so.1.4.5 1769
 28k( 28k): /usr/lib/arm-linux-gnueabihf/gconv/gconv-modules.cache 1880
 136k( 104k): /usr/lib/arm-linux-gnueabihf/ld-2.32.so 1769 1880 2080 2...
 88k( 20k): /usr/lib/arm-linux-gnueabihf/libacl.so.1.1.2253 1769
 116k( 48k): /usr/lib/arm-linux-gnueabihf/libapparmor.so.1.7.0 1769
 92k( 24k): /usr/lib/arm-linux-gnueabihf/libargon2.so.1 1769
```

Утилита atop

Отображает степень использования основных аппаратных ресурсов компьютера. Часто используется системными администраторами, поскольку позволяет рассказать обо всех ресурсах, входящих в состав данной вычислительной системы: о процессорах, об используемых дисках, об устройстве сети, обо всех УВВ.

| ATOP - ubuntu | | | | | | | | | | | | 2021/03/19 17:04:52 | | | |
|---------------|-----------|--------|-------|--------|-------|--------|-------|---------|----------|--------|--------------|---------------------|--------|--------|---|
| | PRC | sys | 0.06s | user | 0.03s | #proc | 137 | #tslpi | 102 | #tslpu | 0 | #zombie | 0 | #exit | 0 |
| CPU | sys | 1% | user | 0% | irq | 0% | idle | 399% | wait | 0% | ipc notavail | avgscal | 50% | | |
| cpu | sys | 0% | user | 0% | irq | 0% | idle | 99% | cpu001 w | 0% | ipc notavail | avgscal | 50% | | |
| cpu | sys | 0% | user | 0% | irq | 0% | idle | 100% | cpu000 w | 0% | ipc notavail | avgscal | 50% | | |
| cpu | sys | 0% | user | 0% | irq | 0% | idle | 100% | cpu003 w | 0% | ipc notavail | avgscal | 50% | | |
| cpu | sys | 0% | user | 0% | irq | 0% | idle | 100% | cpu002 w | 0% | ipc notavail | avgscal | 50% | | |
| CPL | avg1 | 0.12 | avg5 | 0.05 | avg15 | 0.13 | csw | 980 | intr | 3471 | | numcpu | 4 | | |
| MEM | tot | 852.8M | free | 413.7M | cache | 221.6M | buff | 34.9M | slab | 73.4M | vmbal | 0.0M | hptot | 0.0M | |
| SWP | tot | 0.0M | free | 0.0M | | | | | | | vmcom | 333.8M | vmlim | 426.4M | |
| PSI | cs | 0/0/0 | ms | 0/0/0 | mf | 0/0/0 | is | 0/0/0 | if | 0/0/0 | | | | | |
| NET | transport | | tcpi | 9 | tcpo | 14 | udpi | 0 | udpo | 0 | tcpao | 0 | tcppo | 0 | |
| NET | network | | ipi | 10 | ipo | 12 | ipfrw | 0 | deliv | 10 | icmpl | 0 | icmpon | 0 | |
| NET | wlan0 | 0% | pcki | 10 | pcko | 14 | sp | 24 Mbps | si | 0 Kbps | so | 6 Kbps | erro | 0 | |

| PID | SYSCPU | USRCPU | VGROW | RGROW | RUID | EUID | ST | EXC | THR | S | CPUNR | CPU | CMD | 1/1 |
|------|--------|--------|-------|-------|--------|--------|----|-----|-----|---|-------|-----|----------------|-----|
| 2085 | 0.04s | 0.02s | 552K | 1372K | ubuntu | ubuntu | -- | - | 1 | R | 1 | 1% | atop | |
| 2038 | 0.01s | 0.00s | 0K | 0K | root | root | -- | - | 1 | I | 2 | 0% | kworker/u8:2-e | |
| 2082 | 0.01s | 0.00s | 0K | 0K | root | root | -- | - | 1 | I | 0 | 0% | kworker/u8:1-b | |
| 2084 | 0.00s | 0.01s | 0K | 0K | root | root | -- | - | 1 | I | 3 | 0% | kworker/3:2-ev | |
| 1879 | 0.00s | 0.00s | 0K | 0K | ubuntu | ubuntu | -- | - | 1 | S | 3 | 0% | sshd | |
| 2083 | 0.00s | 0.00s | 0K | 0K | root | root | -- | - | 1 | I | 0 | 0% | kworker/u8:3-e | |

Утилита htop

Позволяет отобразить объем резидентной (невыгружаемой памяти), общий объем памяти процесса, размер библиотек, размер страниц памяти и объем памяти для всех запущенных процессов. На экран выводит две таблицы.

| | | |
|------|------------|------------------------------|
| 1 [| 0.0%] | Tasks: 32, 28 thr; 1 running |
| 2 [| 0.0%] | Load average: 0.00 0.02 0.07 |
| 3 [| 1.3%] | Uptime: 00:39:55 |
| 4 [| 0.0%] | |
| Mem[| 150M/853M] | |
| Swp[| 0K/0K] | |

| PID | USER | PRI | NI | VIRT | RES | SHR | S | CPU% | MEM% | TIME+ | Command |
|------|--------|-----|----|-------|-------|-------|---|------|------|---------|-----------------------------------|
| 2274 | ubuntu | 20 | 0 | 5460 | 2568 | 1776 | R | 2.0 | 0.3 | 0:04.62 | htop |
| 1879 | ubuntu | 20 | 0 | 12540 | 3768 | 2840 | S | 0.0 | 0.4 | 0:00.39 | sshd: ubuntu@pts/0 |
| 1501 | root | RT | 0 | 20344 | 14688 | 4920 | S | 0.0 | 1.7 | 0:00.71 | /sbin/multipathd -d -s |
| 1633 | root | 20 | 0 | 15196 | 1376 | 1192 | S | 0.0 | 0.2 | 0:00.59 | /usr/sbin/irqbalance --foreground |
| 1638 | root | 20 | 0 | 18952 | 14092 | 5 | S | 0.0 | 2.2 | 0:27.65 | /usr/lib/snapd/snapd |
| 1962 | root | 20 | 0 | 890M | 18952 | 14092 | S | 0.0 | 2.2 | 0:00.86 | /usr/lib/snapd/snapd |
| 1892 | root | 20 | 0 | 890M | 18952 | 14092 | S | 0.0 | 2.2 | 0:00.09 | /usr/lib/snapd/snapd |
| 1919 | root | 20 | 0 | 890M | 18952 | 14092 | S | 0.0 | 2.2 | 0:01.08 | /usr/lib/snapd/snapd |
| 1505 | root | RT | 0 | 20344 | 14688 | 4920 | S | 0.0 | 1.7 | 0:00.16 | /sbin/multipathd -d -s |
| 877 | root | 19 | -1 | 45880 | 13080 | 12268 | S | 0.0 | 1.5 | 0:00.77 | /lib/systemd/systemd-journald |
| 903 | root | 20 | 0 | 18612 | 3708 | 2792 | S | 0.0 | 0.4 | 0:02.04 | /lib/systemd/systemd-udevd |

В самом верху показана нагрузка на каждое ядро центрального процессора (цифры от 1 до 12).

Mem — это общее количество оперативной памяти и используемая память.

Tasks — обобщённая статистика по процессам.

Swp — уровень занятости файла подкачки (если он есть).

Load average — средняя загрузка центрального процессора.

Uptime — время работы операционной системы с момента последней загрузки.

В нижней таблице программа **htop** выводит следующую информацию (похоже на утилиту **ps** (process status), которая позволяет отслеживать процессы, которые существуют в настоящее время в ОС):

PID — Идентификатор процесса.

USER — Имя пользователя владельца процесса или ID, если имя не может быть определено.

PRI — Приоритет — приоритет процесса, обычно это просто значение NICE плюс двадцать.

NI — NICE значение процесса от 20 (низкий приоритет) до -20 (высокий приоритет). Более высокое значение означает, что процесс «приятный» для других и позволяет им иметь более высокий приоритет выполнения.

VIRT — Размер виртуальной памяти процесса (M_SIZE).

RES — Размер используемой физической памяти.

S — STATE, состояние процесса, может быть:

- **S** для спящих (в простое);
- **R** для запущенных;
- **D** для сна диска (бесперебойный);
- **Z** для зомби (в ожидании чтения статуса своего завершения родительским процессом);
- **T** для отслеживания или приостановки (т.е. от SIGTSTP);
- **W** для подкачки.

CPU% — Процент процессорного времени, которое процесс использует в данный момент.

MEM% — Процент памяти, используемой процессом в данный момент (в зависимости от размера резидентной памяти процесса).

TIME+ — Время, измеренное в часах, указывает на то, сколько процесс провёл в пользовательском и системном времени.

Command — Полная командная строка процесса (то есть имя программы и аргументы).

Возникновение процесса-зомби (англ. *zombie process*, англ. *defunct process*)

Процесс при завершении (как нормальном, так и в результате не обрабатываемого сигнала) освобождает все свои ресурсы и становится «зомби» — пустой записью в таблице процессов, хранящей статус завершения, предназначенный для чтения родительским процессом. Зомби не занимают памяти (как процессы-сироты), но блокируют записи в таблице процессов, размер которой ограничен для каждого пользователя и системы в целом.

Зомби-процесс существует до тех пор, пока родительский процесс не прочитает его статус с помощью *CB wait()*, в результате чего запись в таблице процессов будет освобождена.

При завершении процесса система уведомляет родительский процесс о завершении дочернего с помощью сигнала *SIGCHLD*, таким образом может быть удобно (но не обязательно) осуществлять вызов *wait()* в обработчике данного сигнала.

Процесс-сирота (англ. *orphan process*) — в семействе операционных систем *UNIX* вспомогательный процесс, чей основной процесс (или связь с ним) был завершен нештатно (не подав сигнала на завершение работы).

Обычно, «сиротой» остается дочерний процесс после неожиданного завершения родительского, но возможно возникновение сервера-сироты (локального или сетевого) при неожиданном прерывании связи или завершении клиентского процесса.

Процессы-сироты расходуют системные ресурсы сервера и могут быть источником проблем. Чаще всего такие процессы завершаются посылкой сигнала *SIGTERM* или *SIGKILL*.

Утилита vmstat

Показывает ОП на момент подкачки страницы. Формирует отчет о существующих процессах, распределении виртуальной и физической памяти между существующими процессами, о блок-ориентированных устройствах ввода-вывода, о внешней памяти, и о процессорном времени.

```
ubuntu@ubuntu:~/dim/one$ vmstat
procs      memory          swap--    io-----  system--  -----cpu-----
r b   swpd   free   buff   cache   si   so   bi   bo   in   cs us sy id wa st
0 0     0 363348 36796 324348   0   0   41    7 102   46   1 1 97   2 0
```

Procs — r: Общее количество процессов, ожидающих запуска.

Procs — b: Общее количество занятых процессов.

Memory — swpd: Использованная виртуальная память.

Memory — free: Свободная виртуальная память.

Memory — buff: Память, используемая в качестве буфера.

Memory — cache: Память, используемая в качестве кэша.

Swap — si: Память которая использует swap (swapped) с диска (за каждую секунду).

Swap — so: Память которая использует swap(swapped) на диск (за каждую секунду).

IO — bi: Блоки «in». Т.е. блоки, посылаемые от устройства (за каждую секунду).

IO — bo: Блоки «out». Блоки, посылаемые на устройство (за каждую секунду).

System — in: Количество прерываний в секунду.

System — cs: Переключение контекста.

CPU — us, sy, id, wa, st:

Процессорное время пользователя **us** (CPU user time — % времени CPU, занятый на выполнение «пользовательских», т.е. не принадлежащих ядру задач);

системное время **sy** (CPU system time — % времени CPU, занятый на выполнение задач ядра, т.е. сеть, задачи ввода-вывода, прерывания и т.п.);

время простоя **id** (idle time — % времени в бездействии, т.е. ожидании задач);

время ожидания **wa** (wait time — % времени CPU, занятый на ожидание операций ввода-вывода);

st (Time stolen from a virtual machine — время, позаимствованное из виртуальных машин).

Другие утилиты для работы с оперативной памятью

htop — позволяет определять следующие параметры:

1. ЦПУ;
2. объем памяти;
3. информацию о сети;
4. информацию о дисках;
5. информацию о файловой системе;
6. NFS (Network File System) — информацию о сетевой файловой системе.

smon — позволяет определить:

- PSS (Proportional Set Size) — объем физической памяти, используемой библиотеками и программными приложениями;
- USS (Unique Set Size) — объем свободной памяти;
- RSS (Resident Set Size) — оценку объема физической памяти.

top — позволяет вывести список работающих процессов и информацию о них.

В любой UNIX-подобной ОС существует файл **cat /proc/meminfo** — это автоматически обновляемый файл, позволяющий узнать информацию о распределении ОП.

Не вошедшее (материалы очного курса)

Функционирование ОС UNIX. (7-2, 19-2)

ОС UNIX функционирует как система с разделением времени, предусматривающая одновременную работу нескольких пользователей, каждый из которых обладает терминалом.

Через терминал пользователь общается с системой, передает запросы на выполнение требуемых функций, получает результаты их работы.

Ввод и вывод информации обычно производится на пользовательский терминал, который по умолчанию считается **стандартным вводом и выводом**. Однако при необходимости стандартный ввод и вывод можно переопределить на другие устройства.

Стандартный ввод — файл с дескриптором 0;

Стандартный вывод — файл с дескриптором 1.

Ядро системы создает для каждого пользователя определенную независимую среду выполнения.

При взаимодействии с ОС UNIX пользователь может обращаться к большому количеству файлов.

Настройка среды при входе в систему обеспечивает установку, в качестве текущего, начального каталога пользователя. Начальный каталог определяет поддерево файловой системы, непосредственно доступное пользователю. Тем самым ограничивается набор файлов файловой системы, непосредственно доступных пользователю для изменения.

Диалог в ОС UNIX обеспечивается специальной программой-интерпретатором **shell**, которая составляет неотъемлемую часть среды выполнения каждого пользователя. Интерпретатор **shell** воспринимает команды, посылаемые с терминала, контролирует их и затем, если не обнаруживает ошибок, выполняет. Выполнение отдельной команды может рассматриваться как инициация определенной программы или группы взаимосвязанных программ, обеспечивающих ее реализацию.

Стратегии управления памятью. Концепции распределения памяти. Алгоритмы замещения. (18-1)

Понятие управления ОП включает в себя следующие стратегии (группы алгоритмов):

1. **Стратегия выборки** — ставит целью определить, когда следует втолкнуть (ввести) очередной блок программы или данных в ОП:
 - a) Выборка по запросу;
 - b) Упреждающая выборка.
2. **Стратегия размещения** — ставит целью определить, в какое место следует помещать поступающий блок.
3. **Стратегия замещения** — ставит целью определить, какой блок программы или данных следует вытолкнуть из ОП, чтобы освободить место для записи поступающих программ или данных.

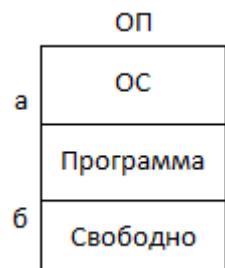
Существует большое количество алгоритмов замещения:

- **Оптимальный алгоритм** — заменяет тот блок, обращение к которому производилось раньше других, находящихся в памяти.
- **FIFO «first-in-first-out» (первый пришел — первый ушел)** — отслеживает порядок загрузки блоков в память, храня их в связном списке. При этом удаление старейшего блока тривиально, но этот блок может быть задействован в данный момент.
- **Вторая попытка** — модификация FIFO, перед удалением блока проверяет, не используется ли он в данный момент и, если используется, то блок пропускается и удаляется следующий.
- **LRU (Least Recently Used)** — удаляет блок, не использовавшийся дольше всех. Требует специального аппаратного обеспечения.
- **Старение (aging)** — программная реализация алгоритма LRU.

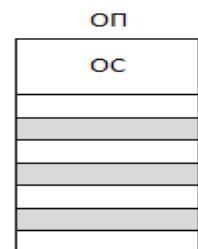
- **NRU (Not Recently Used)** — удаляется блок, не использующийся за последнее время. Используется в ОС Windows.
- **Алгоритм часов** — блоки памяти располагаются как «на циферблате». За первый проход стрелка сбрасывает флаги использования блоков. Если блок используется, то до второго прохода стрелки этот флаг восстановится. Если же до второго прохода стрелки флаг не восстановился, блок считается не используемым и может быть удален. Используется в ОС UNIX. В Mac OS — 2 стрелки. Время между проходами стрелки 20 секунд по умолчанию (устанавливается вручную при генерации системы).

Концепции распределения памяти:

1. **Связное распределение.** При связном распределении программа занимает один сплошной блок ячеек памяти (от а до б). В этом случае размер программы ограничивается количеством памяти. Например, оверлейные перекрытия — программа помещается в память не целиком, а частями.



2. **Несвязное распределение.** При несвязном распределении памяти программа разбивается на ряд блоков или сегментов, которые могут размещаться в ОП на участках, не обязательно соседствующих друг с другом. К несвязному распределению памяти относятся подходы, основанные на свопинге и концепции виртуальной памяти.



Организация памяти при связном распределении. Оверлейные перекрытия. (19-1)

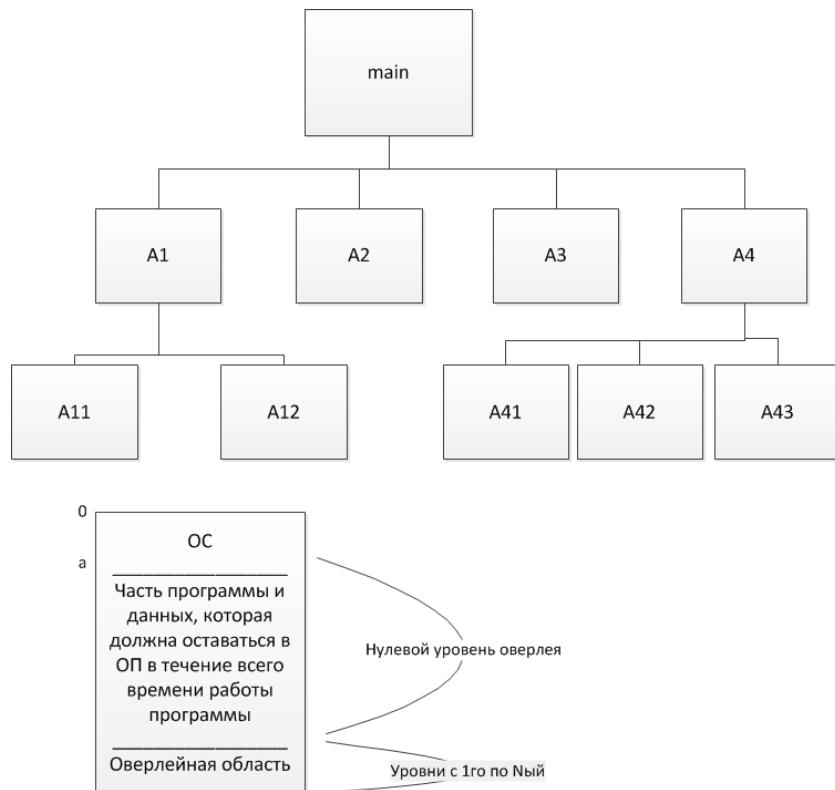
3 типа:

- Программа занимает всю память;
- Постоянные разделы, у каждого своя очередь;
- Переменные разделы, каждой программе выделяется столько памяти, сколько ей требуется.

При связном распределении памяти каждая программа должна занимать один сплошной блок ячеек памяти. В этом случае размер программ ограничивается емкостью имеющейся оперативной памяти.

Однако если объем программы превышает объем имеющейся оперативной памяти, существует возможность её выполнения, благодаря использования оверлейных перекрытий (сегментов).

Суть этого метода заключается в следующем: программа формируется как набор подпрограмм. Т.е. программа делится на части (оверлеи), которым не обязательно находиться в оперативной памяти одновременно.



link a.exe=main / L:0,A1,A11,A12,/L:1,A2,A3/L:2,A4,A41,A42,A43/L:3

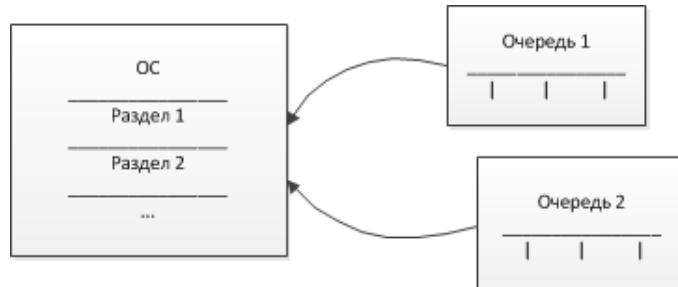
Организация памяти на основе мультипрограммирования. (20-1)

Мультипрограммирование — способ организации выполнения нескольких программ на одном компьютере.

Организация памяти на основе мультипрограммирования с фиксированными разделами.

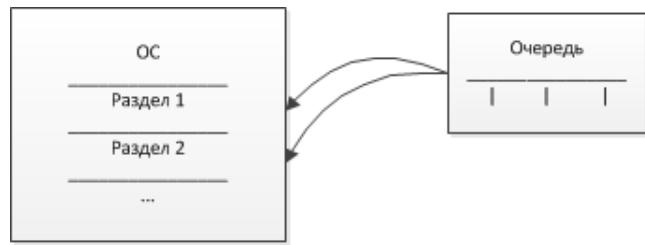
ОП разбивается на ряд разделов фиксированного размера (размеры могут быть разными). В каждом разделе может выполняться одна программа. Формируется очередь программ для каждого из разделов.

ЦП переключается с программы на программу, создавая иллюзию их одновременного выполнения.



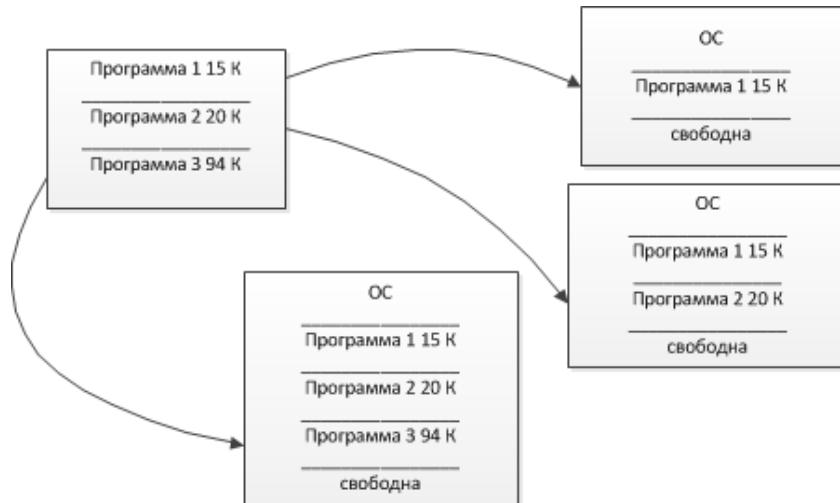
Трансляция программ проводится при помощи ассемблеров и компиляторов в абсолютных адресах с расчетом выполнения только в конкретном разделе.

При использовании перемещающихся компиляторов, ассемблеров и загрузчиков, загрузочные модули можно получать в перемещаемых адресах. Тогда подход трансформируется к следующему виду: существует одна очередь, каждая программа размещается в произвольном свободном разделе.



Организация памяти на основе мультипрограммирования с переменными разделами.

ОП разбивается на ряд разделов переменного размера. Каждой программе выделяется столько памяти, сколько она требует. Закончилась память — очередь ждет, когда память освободится.



Когда программа заканчивает работу, возникает проблема появления свободных участков памяти, часто недостаточно больших для загрузки следующей программы.

В том случае, когда освобождаются смежные разделы, границы между ними удаляются и разделы объединяются. За счет объединения или слияния смежных разделов образуются большие фрагменты, в которых можно разместить большие программы из очереди.

Если же освобождаются несмежные разделы, то в памяти порождается множество малых свободных фрагментов, каждый из которых может быть недостаточен для размещения очередного процесса.

Проблема решается при помощи «уплотнения памяти» или «сбора мусора». Уплотнением памяти называется перемещение всех занятых разделов по адресному пространству памяти таким образом, чтобы свободный фрагмент занимал одну связную область.

Это концепция связного распределения памяти.

Организация памяти на основе свопинга. Битовые карты и связный список свободных и занятых блоков. (21-1)

Свопинг

(обычная подкачка)

Этот подход не требует, чтобы программы оставались постоянно в ОП до момента завершения.

В некоторых системах со свопингом, всю ОП в некоторый момент времени занимает только одна программа. Отработав свой квант времени, эта программа выталкивается из памяти и в память вводится другая программа из очереди.

В обычном случае каждая программа еще до завершения будет многократно перекачиваться из внешней памяти в ОП и наоборот.

В настоящее время разработана система со свопингом, позволяющая размещать в ОП сразу несколько программ.

ОС со свопингом называются ОС с разделением времени.

Битовые карты и связный список свободных и занятых блоков.

Существует два способа учета использования памяти:

- Применение битовых карт.
- Список свободных участков.

Если создаваемый процесс имеет фиксированный размер, размещение его в ОП осуществляется просто: ОС выделяет необходимое количество памяти. Однако, область памяти может расти в результате динамического распределения памяти. Если память выделяется динамически, этим процессом должна управлять ОС, которая использует либо битовые карты, либо список свободных участков.

В **битовой карте** каждому свободному блоку соответствует бит, равный **0**, а каждому занятому — бит, равный **1**.

Другой способ отслеживания состояния памяти — поддержка **связных списков** занятых и свободных блоков. Каждая запись в списке указывает, является ли область памяти свободной, адрес с которого начинается эта область, ее длину, а так же содержит указатель на следующую запись.

Концепция виртуальной памяти. (22-1)

Позволяет программам работать даже тогда, когда они частично находятся в ОП. Основой для разработки подхода послужили оверлейные перекрытия: ОС сама разбивает программу на модули, освобождая программиста от этой работы.

Суть концепции ВП в том, что адреса, к которым обращается выполняющийся процесс, отделяются от адресов реально существующей ОП. Те адреса, на которые делает ссылки выполняющийся процесс, называются **виртуальными**, а адреса ОП — **реальными или физическими**.

Преобразование виртуальных адресов в реальные во время выполнения процесса обеспечивает механизм динамического преобразования адресов (**МДПА**).

МДПА ведет таблицы, показывающие, какие ячейки ВП в текущий момент времени находятся в реальной памяти и где они размещаются.

Чтобы сократить объем информации, отображаемые элементы информации группируются в блоки. Система следит за тем, в каких местах ОП размещаются различные блоки ВП.

Способы реализации ВП

Существует 2 способа реализации виртуальной памяти:

- Если блоки имеют **одинаковый размер**, то они называются страницами, а соответствующая организация памяти — **страничная**;
- Если блоки имеют **различные размеры**, то они называются сегментами, а соответствующая организация памяти — **сегментная**.

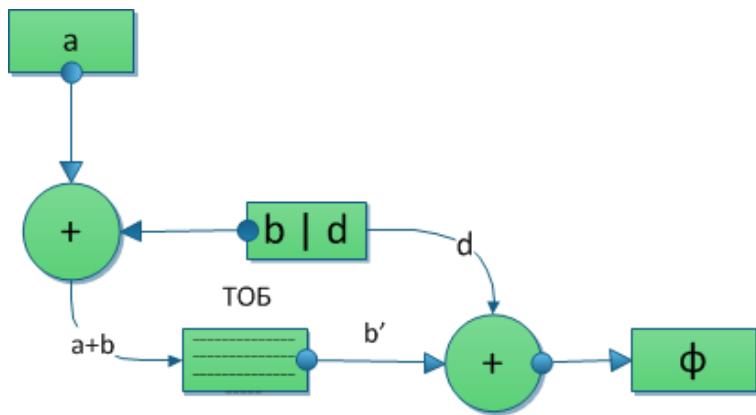
Преобразование V в Ψ

Ψ — физический адрес в ОП.

V — виртуальный адрес, указывается при помощи упорядоченной пары чисел **(b,d)**, где **b** — номер блока; **d** — смещение, относительно начального адреса.

Каждый процесс имеет собственную таблицу отображения блоков (ТОБ = Таблица Страниц — отображает виртуальные страницы в реальные), которую ОС ведет в реальной физической памяти (**Ψ**).

Реальный адрес «**a**» в этой таблице загружается в специальный регистр ЦП, называемый регистром начального адреса таблицы блоков (РНАТБ). Номер блока «**b**» суммируется с базовым адресом «**a**», образуя реальный адрес «**b'**» блока «**b**» в физической памяти. Затем к «**b'**» прибавляется смещение «**d**» и получается **Ψ** — реальный адрес.



Из-за этого систему нельзя назвать системой реального времени.

Расслоение памяти. Регистр перемещений. Прерывания и опрос состояний. (2-1)

Метод расслоения памяти (интерликинг)

Применяется для увеличения скорости доступа к основной (оперативной) памяти. В обычном случае, во время обращения к какой-то одной из ячеек модуля основной памяти, никакие другие обращения к памяти производиться не могут. При расслоении памяти соседние по адресам ячейки размещаются в различных модулях памяти, так что появляется возможность производить несколько обращений одновременно.

Например, при расслоении на два направления, ячейки с нечетными адресами оказываются в одном модуле памяти, а с четными — в другом. При простых последовательных обращениях к основной памяти ячейки выбираются поочередно.

Таким образом, расслоение памяти позволяет обращаться сразу к нескольким ячейкам, поскольку они относятся к различным модулям памяти.

Регистр перемещения

Регистр перемещения обеспечивает возможность динамического перемещения программ в памяти.

В регистр перемещения заносится базовый адрес программы, хранящейся в основной памяти. Содержимое регистра перемещения прибавляется к каждому указанному в выполняемой программе адресу. Благодаря этому, пользователь может писать программу так, как если бы она начиналась с нулевой ячейки памяти.

Во время выполнения программы все исполнительные адреса обращений формируются с использованием регистра перемещения — и благодаря этому программа может реально размещаться в памяти совсем не в тех местах, которые она должна была занимать согласно адресам, указанным при трансляции.

Прерывания и опрос состояний

Одним из способов, позволяющих некоторому устройству проверить состояние другого, независимо работающего устройства, является **опрос**. Например, первое устройство может периодически проверять, находится ли второе устройство в определенном состоянии, и, если нет, то продолжать свою работу. Опрос может быть связан с высокими накладными расходами.

Прерывания дают возможность одному устройству немедленно привлечь внимание другого устройства, с тем, чтобы первое могло сообщить об изменении своего состояния.

Состояние устройства, работа которого прерывается, должно быть сохранено, только после этого можно производить обработку данного прерывания. После завершения обработки прерывания, состояние прерванного устройства восстанавливается для того, чтобы можно было продолжить работу.

Буферизация. Периферийные устройства. Защита памяти. (3-1)

Буферизация

Буфер — это область памяти, предназначенная для промежуточного хранения данных при выполнении операций ввода-вывода. Скорость выполнения операции ввода-вывода зависит от многих факторов, связанных с характеристиками аппаратуры ввода-вывода, однако в обычном случае ввод-вывод производится не синхронно с работой процессора. Например, при вводе, данные помещаются в буфер средствами канала ввода-вывода и, после занесения данных в буфер, процессор получает возможность доступа к этим данным.

При вводе с **простой буферизацией** канал помещает данные в буфер, процессор обрабатывает эти данные, канал помещает следующие данные и т.д. В то время, когда канал заносит данные, обработка этих данных производиться не может, а во время обработки данных нельзя заносить дополнительные данные.

Метод **двойной буферизации** позволяет совмещать выполнение операции ввода-вывода с обработкой данных; когда канал заносит данные в один буфер, процессор может обрабатывать данные другого буфера. А когда процессор заканчивает обработку данных одного буфера, канал будет заносить новые данные опять в первый буфер.

Периферийные устройства

ЦП, память и блок питания — компьютер. Все остальное — периферийные устройства.

Периферийное устройство — часть технического обеспечения, конструктивно отделенная от основного блока вычислительной системы. Периферийные устройства имеют собственное управление и функционируют по командам центрального процессора.

Периферийные устройства предназначены для внешней обработки данных, обеспечивают их подготовку, ввод, хранение, управление, защиту, вывод и передачу на расстояние по каналам связи. Примеры периферийных устройств: жесткий диск, CD-ROM, клавиатура, принтер, монитор и т.д.

Защита памяти

Защита памяти ограничивает диапазон адресов, к которым разрешается обращаться программе.

Защиту памяти для программы, занимающей непрерывный блок ячеек памяти, можно реализовать при помощи так называемых **границных регистров**, где указываются старший и младший адреса этого блока памяти. При выполнении программы все адреса обращения к памяти контролируются, чтобы убедиться в том, что они находятся в промежутке между адресами, указанными в границных регистрах.

Защиту памяти можно реализовать также при помощи **ключей защиты памяти**, относящихся к определенным областям основной памяти. Программе разрешается обращение к ячейкам памяти только тех областей, ключи которых совпадают с ключом данной программы.

Таймер. Каналы ввода-вывода. Захват цикла. (4-1)

Таймер и часы

Интервальный таймер — эффективный способ предотвращения монополизации процессора одним процессором. По истечении заданного интервала времени, таймер генерирует сигнал прерывания для привлечения внимания процессора. По этому сигналу процессор может переключиться на обслуживание другого процесса.

Часы истинного времени дают возможность компьютеру следить за реальным календарным временем с точностью до миллионных долей секунды, а при необходимости даже точнее.

Каналы ввода-вывода

Канал ввода-вывода — это специализированный процессор, предназначенный для управления вводом-выводом независимо от основного процессора вычислительной машины. Канал имеет возможность прямого доступа к основной памяти для записи или выборки информации.

В современных машинах с управлением по прерываниям процессор выполняет команду «начать ввод-вывод» (SIO), чтобы инициировать передачу данных ввода-вывода по каналу. После окончания операции ввода-вывода, канал выдает сигнал прерывания, уведомляющий процессор о событии завершения операции ввода-вывода.

Истинное значение каналов состоит в том, что они позволяют значительно увеличить параллелизм работы аппаратуры компьютера и освобождают процессор от подавляющей части нагрузки, связанной с управлением вводом-выводом.

- Для высокоскоростного обмена данными между внешними устройствами и основной памятью используется **селекторный канал**. Селекторные каналы имеют только по одному подканалу и могут обслуживать только одно устройство в каждый момент времени.
- **Мультиплексные каналы** имеют много подканалов, они могут работать сразу со многими потоками данных в режиме чередования.
- **Байт-мультиплексный канал** обеспечивает режим чередования байтов при одновременном обслуживании ряда таких медленных внешних устройств, как терминалы, перфокарточные устройства ввода-вывода, принтеры, а также низкоскоростные линии передачи данных.
- **Блок-мультиплексный канал** при обмене в режиме чередования блоков может обслуживать несколько таких высокоскоростных устройств, как лазерные принтеры и дисковые накопители.

Захват цикла

Узкое место, в котором может возникнуть конфликтная ситуация между каналами и процессором — это доступ к основной памяти. Поскольку в каждый конкретный момент времени может осуществляться только одна операция обращения (к некоторому модулю основной памяти) и, поскольку каналам и процессору может одновременно потребоваться обращение к основной памяти, то в обычном случае приоритет здесь предоставляется каналам. Это называется захватом цикла памяти, при котором канал буквально захватывает, или «крадет» циклы обращения к памяти у процессора. Каналам требуется лишь небольшой процент общего числа циклов памяти, а предоставление им приоритета в этом смысле позволяет обеспечить лучшее использование устройств ввода-вывода. Подобный подход принят и в современных операционных системах: планировщики, входящие в состав операционной системы, как правило, отдают приоритет программам с большим объемом ввода-вывода по отношению к программам с большим объемом вычислений.

Относительная адресация. Режимы работы ЭВМ. (5-1)

Относительная адресация

Когда потребность в увеличении объемов основной памяти стала очевидной, архитектуры компьютеров были модифицированы для работы с очень большим диапазоном адресов. Машина, рассчитанная на работу с памятью емкостью 16 МБ (1 МБ — это 1 048 576 байт), должна иметь 24-разрядные адреса. Включение столь длинных адресов в формат каждой команды даже для машины с одноадресными командами стоило бы очень дорого, не говоря уже о машинах с многоадресными командами.

Поэтому для обеспечения работы с очень большими адресными пространствами в машинах начали применять адресацию типа база + смещение, или **относительную адресацию**, при которой все адреса программы суммируются с содержимым **базового регистра**. Подобное решение имеет дополнительное преимущество, при котором программы становятся **перемещаемыми**, или **позиционно-независимыми**.

Режим работы компьютера

В вычислительных машинах, как правило, предусматривается несколько различных **режимов работы**. Динамический выбор режима позволяет лучше организовать защиту программ и данных.

В обычном случае, когда машина находится в конкретном режиме, работающая программа может выполнять только некоторое подмножество команд. В частности, если говорить о программах пользователя, то подмножество команд, которые пользователь может употреблять в **режиме задачи**, не позволяет,

например, непосредственно производить операции ввода-вывода. Если программе пользователя разрешить выполнение любых операций ввода-вывода, то она могла бы вывести главный список паролей системы, распечатать информацию любого другого пользователя или вообще испортить операционную систему.

Операционной системе обычно присваивается статус самого полномочного пользователя и работает она в режиме супервизора; она имеет доступ ко всем командам, предусмотренным в машине.

Для большинства современных вычислительных машин подобного разделения на два режима — задачи и супервизора — вполне достаточно. Однако, в случае машин с высокими требованиями по защите от несанкционированного доступа, желательно иметь более двух режимов.

Виртуальная память. Мультипроцессорная обработка. Прямой доступ к памяти. (6-1)

Виртуальная память

Системы виртуальной памяти дают возможность указывать в программах адреса, которым не обязательно соответствуют физические адреса основной памяти. Виртуальные адреса, выдаваемые работающими программами, при помощи аппаратных средств динамически (т.е. во время выполнения программы) преобразуются в адреса команд и данных, хранящихся в основной памяти. Системы виртуальной памяти позволяют программам работать с адресными пространствами гораздо большего размера, чем адресное пространство основной памяти. Они дают пользователям возможность создавать программы, независимые (большей частью) от ограничений основной памяти, и позволяют обеспечить работу многоабонентских систем с общими ресурсами.

В системах виртуальной памяти применяются такие методы, как **страничная организация**, которая предусматривает обмен между основной и внешней памятью блоками данных фиксированного размера, и **сегментация**, которая предусматривает разделение программ и данных на логические компоненты, называемые сегментами, что упрощает управление доступом и коллективное использование. Эти методы иногда реализуются порознь, а иногда в комбинации.

Мультипроцессорная обработка

В **мультипроцессорных** машинах несколько процессоров работают с общей основной памятью и одной операционной системой. При мультипроцессорной работе возникает опасность конфликтных ситуаций определенных типов, которых не бывает в однопроцессорных машинах. Здесь необходимо обеспечить координированный **упорядоченный** доступ к каждой общей ячейке памяти, с тем чтобы два процессора не могли изменять ее содержимое одновременно — и в результате, быть может, портить его. Подобная координация необходима также и в случае, когда один процессор пытается изменить содержимое ячейки, которую хочет прочитать другой процессор. Упорядочение доступа необходимо также и для однопроцессорных машин.

Прямой доступ к памяти

(По сути эквивалентен каналу ввода-вывода)

Одним из способов достижения высокой производительности вычислительных машин является минимизация количества прерываний, происходящих в процессе выполнения программы. Разработанный для этого способ **прямого доступа к памяти** (ПДП) требует лишь одного прерывания на каждый блок символов, передаваемых во время операции ввода-вывода. Благодаря этому, обмен данными производится значительно быстрее, чем в случае, когда процессор прерывается при передаче каждого символа.

После начала операции ввода-вывода, символы передаются в основную память по принципу захвата цикла — канал захватывает шину связи процессора с основной памятью на короткое время передачи одного символа, после чего процессор продолжает работу.

Когда внешнее устройство оказывается готовым к передаче очередного символа блока, оно «прерывает» процессор. Однако, в случае ПДП, состояние процессора запоминать не требуется, поскольку передача

одного символа означает для процессора скорее задержку, или приостановку, чем обычное прерывание. Символ передается в основную память под управлением специальных аппаратных средств, а после завершения передачи процессор возобновляет работу.

Аппаратные средства, обеспечивающие захват циклов памяти и управление устройствами ввода-вывода в режиме ПДП, называются **каналом прямого доступа к памяти**.

Программирование на машинном языке. Ассемблеры и макропроцессоры. Компиляторы. (7-1)

Программирование на машинном языке

Машинный язык — это язык программирования, непосредственно воспринимаемый компьютером.

Каждая команда машинного языка интерпретируется аппаратурой, выполняющей указанные функции.

Команды машинного языка в принципе являются довольно примитивными. Только соответствующее

объединение этих команд в программы на машинном языке дает возможность описывать достаточно

серьезные алгоритмы. Наборы команд машинного языка современных компьютеров зачастую включают

некоторые очень эффективные возможности.

Говорят, что **машинный язык является машинно-зависимым**: программа, написанная на машинном языке компьютера одного типа, как правило, не может выполняться на компьютере другого типа, если его **машинный язык не идентичен машинному языку первого компьютера (или не является расширением по отношению к этому языку)**. Еще одним признаком машинной, или аппаратной, зависимости является **характер самих команд**: в командах машинного языка указываются наименования конкретных регистров компьютера и предусматривается обработка данных в той физической форме, в которой они существуют в этом **компьютере**. Большинство первых компьютеров программировались непосредственно на машинном языке, а в настоящее время на машинном языке пишется лишь очень небольшое число программ.

Ассемблеры и макропроцессоры

Программирование на машинном языке требует много времени и чревато ошибками. Поэтому были разработаны **языки ассемблерного типа**, позволяющие повысить скорость процесса программирования и уменьшить количество ошибок кодирования. Вместо чисел, используемых при написании программ на **машинных языках**, в языках ассемблерного типа применяются **содержательные мнемонические сокращения и слова естественного языка**. Однако компьютеры не могут непосредственно воспринять программу на языке ассемблера, поэтому ее необходимо вначале перевести на машинный язык. Такой перевод осуществляется при помощи программы-транслятора, называемой **ассемблером**.

Языки ассемблерного типа также являются машинно- зависимыми. Их команды прямо и однозначно соответствуют командам программы на машинном языке. Чтобы ускорить процесс кодирования программы на языке ассемблера, были разработаны и включены в ассемблеры так называемые **макропроцессоры**. Программист пишет **макрокоманду**, как указание необходимости выполнить действие, описываемое несколькими командами на языке ассемблера. Когда макропроцессор во время трансляции программы читает макрокоманду, он производит **макрорасширение** — т.е. генерирует ряд команд языка ассемблера, соответствующих данной макрокоманде. Таким образом, процесс программирования значительно ускоряется, поскольку программисту приходится писать меньшее число команд для определения того же самого алгоритма.

Ассемблер переводит в машинный код за один этап;

Компилятор — за несколько: сначала в ассемблер, затем в машинный код.

Компиляторы

Тенденция к повышению вычислительной мощности команд привела к разработке некоторых очень эффективных макропроцессоров и макроязыков, упрощающих программирование на языке ассемблера.

Однако развитие ассемблеров путем введения макропроцессоров не решает проблемы машинной зависимости. В связи с этим были разработаны **языки высокого уровня**.

Языки высокого уровня открывают возможность **машино-независимого** программирования.

Большинству пользователей компьютер нужен только как средство реализации прикладных систем. Языки высокого уровня позволяют пользователям заниматься преимущественно задачами, специфичными для их конкретных прикладных областей, не вникая в особенности применяемых ими машин.

Перевод с языков высокого уровня на машинный язык осуществляется при помощи программ, называемых **компиляторами**. Компиляторы и ассемблеры имеют общее название «**трансляторы**». Написанная пользователем программа, которая в процессе трансляции поступает на вход транслятора, называется **исходной программой**; программа на машинном языке, генерируемая транслятором, называется **объектной программой**, или **выходной** (целевой) программой.

Конвейеризация. Иерархия памяти. (8-1)

Конвейеризация

Конвейеризация — это аппаратный способ, применяемый в высокопроизводительных вычислительных машинах с целью использования определенных типов параллелизма для повышения эффективности обработки команд. Упрощенно структуру конвейерного процессора можно представить очень похожей на технологическую линию производственного предприятия; на конвейере процессора на различных стадиях выполнения одновременно могут находиться несколько команд. Такое совмещение требует несколько большего объема аппаратуры, однако позволяет существенно сократить общее время выполнения последовательности команд.

Иерархия памяти

Современные вычислительные машины содержат несколько видов памяти:

- **основную** (первичную, оперативную);
- **внешнюю** (вторичную, массовую);
- **кэш-память**.

В **основной памяти** должны размещаться команды и данные, к которым будет обращаться работающая программа.

Внешняя память — это магнитные ленты, диски, карты и другие носители, предназначенные для хранения информации, которая со временем будет записана в основную память.

Кэш-память — это буферная память очень высокого быстродействия, предназначенная для повышения скорости выполнения работающих программ. Для программ пользователя эта память, как правило, «прозрачна».

Все эти виды памяти создают единую **иерархию памяти**. Переход по уровням этой иерархии от кэш-памяти к основной и затем к внешней памяти сопровождается уменьшением стоимости и скорости и увеличением емкости памяти.

Система управления вводом-выводом. Спулинг. (10-1)

Система управления вводом-выводом

Одна из важнейших функций ядра ОС состоит в **управлении устройствами ввода-вывода** компьютера.

Ядро ОС должно давать этим устройствам команды, перехватывать прерывания, обрабатывать ошибки и обеспечивать интерфейс с остальной частью ОС.

Устройства ввода-вывода делятся на:

1. **Блочные устройства** — хранят информацию в виде адресуемых блоков фиксированного размера.

Обычно размеры блока меняются в пределах от одного сектора диска (512 байт) до одного цилиндра диска (32 768 байт). Важное свойство блочного устройства — **каждый блок может быть прочитан**

независимо от остальных блоков. Пример таких устройств — дисковые накопители, информация в которых передается блоками.

2. **Байтовые устройства** — принимают или предоставляют поток байтов или **символов без какой-либо структуры. Не являются адресуемыми.** Пример такого устройства — клавиатура, информация с которой передается в виде байтов.

Программа пользователя, как правило, общается с абстрактными устройствами. Зависимую от устройства часть поддерживает ОС, т.е. драйверы устройств входят в ядро.

Устройства ввода-вывода состоят из механической и электронной частей. Механическая компонента находится в самом устройстве, а электронная (контроллер устройства) принимает форму печатной платы, которая сопряжена с системной шиной или вставляется в слот расширения объединенной платы.

Работа контроллера состоит в последовательном преобразовании потока битов в байтовую последовательность и в выполнении коррекции ошибок. У каждого контроллера есть несколько **регистров (или портов)**, с помощью которых к нему может обращаться ЦП. У некоторых компьютеров (Motorola) такие регистры являются частью единого адресного пространства (ОП). У других (IBM PC) для этого отводится специальное адресное пространство, в котором выделяются адреса для каждого устройства.

В дополнение к регистрам ввода-вывода часто используются **прерывания**, при помощи которых контроллер может сообщить ЦП, что его регистры готовы для записи или чтения информации.

Прерывание является электрическим сигналом. **Линия запроса аппаратного прерывания (IRQ — Interrupt Request Line)** является одним из входов контроллера. Каждая из линий IRQ связывается с **вектором прерываний**, который указывает на программу обработки прерываний.

ОС обменивается с устройством ввода-вывода информацией, записывая команды в регистры контроллера. Передав команду контроллеру, процессор инициирует прерывание, чтобы привлечь внимание ОС для проверки результата.

ПО системы управления вводом-выводом

Оно разрабатывается исходя из принципов независимости от устройств, единообразия именования устройств и многослойности. ПО должно аккумулировать и обрабатывать ошибки устройств ввода-вывода, поддерживать **синхронный (блокирующий) и асинхронный способы переноса данных**, предусматривать буферизацию и различать **выделенные (монопольные) устройства ввода-вывода и устройства ввода-вывода коллективного пользования**.

Уровни (слои) ПО можно представить следующим образом:

1. Обработчик прерываний (нижний слой);
2. Драйвер устройства ввода-вывода;
3. Независимый от аппаратуры программный модуль ОС;
4. Пользовательские программы (верхний слой).

Пункты 1-3 — функции ОС.

Спулинг

При **спулинге** (вводе-выводе с буферизацией) посредником между работающей программой и низкоскоростным устройством, осуществляющим ввод-вывод данных для этой программы, становится высокоскоростное устройство, например дисковый накопитель. Вместо вывода строк данных непосредственно, скажем, на построчно печатающее устройство, программа записывает их на диск. Благодаря этому текущая программа может быстрее завершиться для того, чтобы другие программы могли быстрее начать работать. А записанные на диск строки данных можно распечатать позже, когда освободится принтер.

Процедурно-ориентированные и проблемно-ориентированные языки. Интерпретаторы. (11-1)

Процедурно-ориентированные и проблемно-ориентированные языки.

ПРИМЕРЫ!!

Языки высокого уровня бывают либо процедурно-ориентированными, либо проблемно-ориентированными.

Процедурно-ориентированные языки высокого уровня — это универсальные языки программирования, которые можно использовать для решения самых разнообразных задач.

Проблемно-ориентированные языки предназначаются специально для решения задач конкретных типов.

Интерпретаторы

Существует один интересный и распространенный вид трансляторов, **интерпретаторы**, которые не генерируют объектную программу, а фактически **обеспечивают непосредственное выполнение исходной программы**. Интерпретаторы особенно распространены в системах проектирования программ, где программы идут, как правило, лишь небольшое время до момента обнаружения очередной ошибки. Интерпретаторы распространены также в сфере персональных компьютеров. Они **свободны от накладных затрат, свойственных ассемблированию или компиляции**. Однако **выполнение программы в режиме интерпретации идет медленно** по сравнению с компилированным кодом, **поскольку интерпретаторам надо транслировать каждую команду при каждом ее выполнении**.

Абсолютные и перемещающие загрузчики. Связывающие загрузчики и редакторы связей. (12-1)

Абсолютные и перемещающие загрузчики

Программы для выполнения должны размещаться в основной памяти. Распределение команд и элементов данных по конкретным ячейкам основной памяти является исключительно важной задачей. Решение этой задачи иногда осуществляет сам пользователь, иногда транслятор, иногда системная программа, называемая **загрузчиком**, а иногда — операционная система.

Сопоставление команд и элементов данных с конкретными ячейками памяти называется привязкой к памяти. При программировании на машинном языке привязка к памяти производится в момент кодирования. Уже давно наблюдается тенденция откладывать привязку программы к памяти на как можно более поздний срок, и в современных системах виртуальной памяти привязка осуществляется динамически в процессе выполнения программы.

Загрузчик — это системная программа, которая размещает команды и данные программы в ячейках основной памяти.

- **Абсолютный загрузчик** размещает эти элементы именно в те ячейки, адреса которых указаны в программе на машинном языке.
- **Перемещающий загрузчик** может загружать программу в различные места основной памяти в зависимости, например, от наличия свободного участка основной памяти в момент загрузки.

Связывающие загрузчики и редакторы связей

В настоящее время программы пользователя зачастую содержат лишь незначительную часть команд и данных, необходимых для решения поставленной задачи. В составе системного программного обеспечения поставляются большие **библиотеки подпрограмм**, так что программист, которому необходимо выполнять определенные стандартные операции, может воспользоваться для этого готовыми подпрограммами. В частности, операции ввода-вывода обычно выполняются подпрограммами, находящимися вне программы пользователя. Поэтому программу на машинном языке, полученную в результате трансляции, приходится, как

правило, комбинировать с другими программами на машинном языке, чтобы сформировать необходимый выполняемый модуль.

Эту процедуру объединения программ выполняют **связывающие загрузчики и редакторы связей** до начала выполнения программы.

- Связывающий загрузчик во время загрузки объединяет необходимые программы и загружает их непосредственно в основную память.
- Редактор связей также осуществляет подобное объединение программ, однако он создает **загрузочный модуль**, который записывается во внешнюю память для будущего использования.

Микропрограммирование. Эмуляция. Горизонтальный и вертикальный микрокод. (13-1)

Микропрограммы

Микропрограммирование вводит дополнительный уровень средств программирования, нижележащий по отношению к машинному языку компьютера, и тем самым оно позволяет определять конкретные команды машинного языка. Подобные возможности являются неотъемлемой частью архитектуры современных компьютеров и имеют громадное значение с точки зрения обеспечения высоких скоростных характеристик и защиты операционных систем.

Микропрограммы размещаются в специальной управляющей памяти очень высокого быстродействия. Они состоят из индивидуальных микрокоманд, которые гораздо более элементарны по своей природе и более рассредоточены по функциям, чем обычные команды машинного языка. В компьютерах, где набор команд машинного языка реализуется при помощи микропрограммирования, каждой команде машинного языка соответствует целая и, быть может, большая микропрограмма.

Горизонтальный и вертикальный микрокод

Команды микрокода можно разделить на **горизонтальные и вертикальные**.

Выполнение вертикальных микрокоманд очень похоже на выполнение обычных команд машинного языка. Типичная вертикальная микрокоманда задает пересылку одного или нескольких элементов данных между регистрами.

Горизонтальный микрокод действует совсем по-другому. Каждая его команда содержит гораздо большее число бит, поскольку может задавать параллельную операцию пересылки данных для многих или даже всех регистров данных устройства управления. Горизонтальные микрокоманды являются более мощными, чем вертикальные, однако может оказаться, что соответствующие микропрограммы гораздо сложнее кодировать и отлаживать.

Эмуляция

Эмуляция — метод, позволяющий сделать одну вычислительную машину функциональным эквивалентом другой.

Набор команд машинного языка эмулируемого компьютера микропрограммируется на **эмулирующем компьютере** — и благодаря этому программы, представленные на машинном языке первого компьютера, могут непосредственно выполняться на втором. Фирмы-разработчики компьютеров широко применяют эмуляцию при внедрении новых машин, и пользователи, привязанные к старым компьютерам, получают, например, возможность без всяких изменений выполнять свои ранее отлаженные программы на новых машинах.

Операционная система UNIX

Начальная загрузка и выход на интерактивный режим в ОС UNIX. (25-2)

Для выполнения начальной загрузки необходим диск, содержащий в нулевом блоке **программу-загрузчик**. Эта программа, при соответствующей инициации (обычно аппаратными средствами), считывается

в память. Работа этой программы заключается в отыскании и чтении в ОП ядра ОС. Сначала загрузчик копирует в себя фиксированный адрес памяти в старших адресах, чтобы освободить нижнюю память для ОС. Загрузившись, он считывает корневой каталог с диска. Затем считывает ядро ОС и передает ему управление. Начальная программа ядра устанавливает указатель стека, определяет тип ЦП, вычисляет количество имеющейся в наличии ОП, запрещает прерывания, разрешает работу диспетчера процессов и запускает основную процедуру инициации ОС. **Процесс загрузки разбивается на две фазы:**

- **инициализация ядра;**
- **инициализация системы.**

Инициализация ядра начинается с установки таймера и формирования начальных значений структуры данных ядра. На следующем шаге ОС считывает файлы конфигурации, в которых сообщается, какие типы устройств могут присутствовать, и проверяет, какие из устройств действительно присутствуют. Далее ищутся и подгружаются драйверы действительно присутствующих устройств. Затем ядро специальным образом **без использования СВ fork()** запускает процесс с PID = 0 (swapper). Для него создается контекст, и информация заносится в таблицу процессов.

Процесс с PID = 0 является системным процессом т.к. он постоянно находится в основной памяти и активен только тогда, когда процессор находится в состоянии «система». Процесс с PID = 0 продолжает инициализацию, осуществляя монтирование основной файловой системы. После образования процесса с PID = 0, ядро создает процесс с идентификатором PID = 1 (init) и идентификатором PID = 2 (page demon — страничный демон). Процесс с PID = 1 создается по упрощенной схеме, т.е. с использованием СВ fork(), и служит для загрузки в основную память программы из файла /etc/init. Ядро по системному вызову exec() считывает в процедурный сегмент процесса с PID = 1 программу, которая в будущем обеспечит активизацию из файла /etc/init. На этом инициализация ядра завершается.

Инициализация системы начинается с передачи управления диспетчеру процессов, который выбирает единственный активный процесс с идентификатором PID = 1 и передает управление программе, размещенной в его процедурном сегменте. Процесс с PID = 1 по системному вызову exec() на место своего процедурного сегмента считывает программу /etc/init. Она может создавать два типа режимов организации процессов:

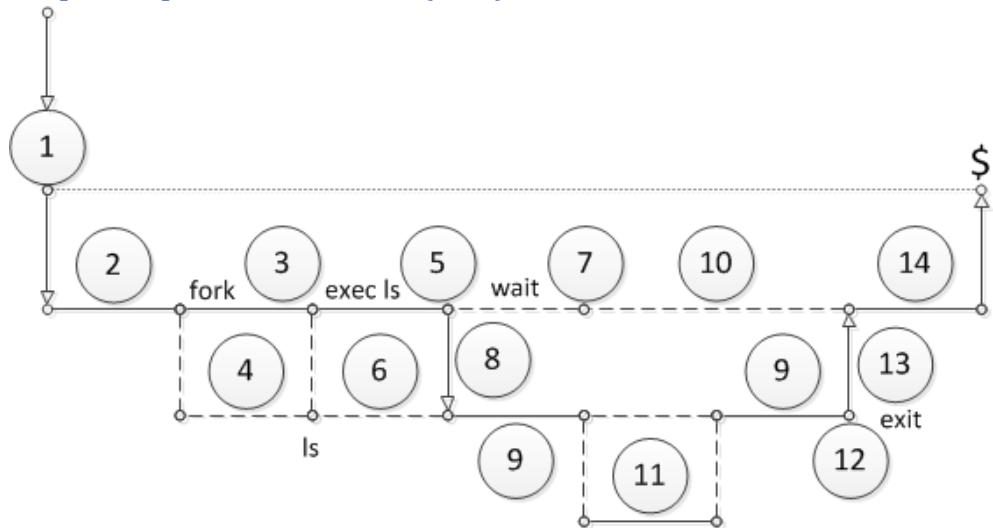
- **однопользовательский;**
- **многопользовательский.**

Процесс init прежде всего инициирует начало работы интерпретатора shell для привилегированного пользователя. При выходе из однопользовательского режима создается многопользовательская среда. Активизация shell делает его доступным для всех пользователей системы. Для каждого активного канала связи порождается процесс getty (сокращение от get teletype) — программа для UNIX-подобных ОС, управляющая доступом к физическим и виртуальным терминалам (tty). Программа выполняет запрос имени пользователя и запускает программу “login” для авторизации пользователя. При вводе в систему по какому-либо каналу процесс getty выдает сообщение:

«Имя (login) и пароль (password)»

Getty может быть использована системными администраторами для предоставления доступа к другим программам. Например, для предоставления доступа к демону pppd для получения dial-up интернет-соединения.

Интерактивный режим работы в ОС UNIX. (24-2)



1. Входим в систему и набираем имя команды **ls** и нажимаем enter
2. Далее управление передается интерпретатору shell, который анализирует введенный текст. Если в этот момент будет зафиксирована ошибка, то интерпретатор проинформирует об этом пользователя, выдав соответствующее сообщение, и снова вернет приглашение для ввода командной строки.
3. Если команда была задана корректно, то shell по средствам системного вызова **fork()** порождает новый процесс.
4. Ядром системы выполняются стандартные действия, в том числе в таблицу процессов заносится запись о новом процессе.
5. Затем из shell выполняется системный вызов **exec()**. В каталоге /bin/ отыскивается файл с именем **ls** и загружается в основную память. Сразу после порождения нового процесса (4) они оба (и процесс-отец shell, и процесс-сын **ls**) начинают конкурентную борьбу за ресурсы системы. Но shell, будучи особым процессом, порождает другие процессы с меньшим приоритетом. Поэтому после порождения процесса управление вернется в shell, и он сможет выполнить системный вызов **exec()**.
6. После его выполнения уже нельзя с определенностью сказать, какой процесс (shell или **ls**) получит управление.
7. Если окажется, что shell, то его необходимо приостановить, чтобы дать возможность выполниться процессу **ls**. Приостановка shell достигается с помощью системного вызова **wait()**. В этой точке выполнение процесса shell прекратится, и он будет ожидать завершения работы процесса **ls**. После остановки shell начинает работу ядро ОС. Диспетчер процессов определяет процесс, который надо активизировать следующим.
8. Т.к. в нашем случае других готовых к выполнению процессов кроме **ls** нет, то ему и будет передано управление.
9. Процесс **ls** начинает работу по выполнению своих функций.
10. В это время процесс shell будет находиться в состоянии ожидания. При выполнении процесса **ls** ему надо будет выводить результаты своей работы на экран терминала.
11. Т.к. в ОС UNIX все программы выполняются последовательно, для их параллельного выполнения требуется создание новых процессов. Для вывода данных **ls** на терминал порождается новый процесс.
12. Выполнение процесса **ls** завершается системным вызовом **exit()**.
13. Ядро системы по взаимодействию пары системных вызовов **wait()** и **exit()** переводит процесс shell в состояние готовности. Ввиду отсутствия других готовых процессов, активизирует его.
14. Интерпретатор shell выполняет завершающее действие по обработке **ls** и выдает на экран терминала приглашение для ввода следующей команды.

13. Интерпретатор команд shell в ОС UNIX. Функции интерпретатора. Встроенные и внешние команды (утилиты) интерпретатора shell

13.1 Интерпретатор команд shell. Версии. Функции

Интерпретатор команд shell — одна из важнейших программ ОС UNIX. Присутствует во всех UNIX-подобных ОС. Разработано достаточно много версий этой программы (более 90), но все они **обеспечивают интерфейс «пользователь-ядро»**.

Интерпретатор команд shell. Версии

На современном этапе наиболее распространенные версии интерпретатора shell следующие:

- **B – shell** (автор Stephen Bourne, Bell Laboratories AT & T, наиболее популярен в настоящее время, классика);
- **C – shell** (автор Bill Joy, Калифорнийский университет, С-подобный синтаксис);
- **K – shell** (автор David Korn, Bell Labs AT & T, обобщены возможности B-shell и C-shell, применяется в ОС UNIX семейства AIX (аббр. от англ. Advanced Interactive eXecutive) корпорации IBM);
- **P – shell** (основан на стандарте POSIX — Portable Operating System Interface, разработан на базе предыдущих разработок B-shell, C-shell и K-shell, подобен K-shell);
- **Z – shell** (Paul Falstad, Принстонский университет, похож на B-Shell, открытый проект).

Интерпретатор shell. Функции

Функции интерпретатора:

1. **Интерпретация команд.**
2. **Обработка имен файлов, которые определены через метасимволы** (в именах файлах могут встречаться различные метасимволы, которые интерпретатор shell должен интерпретировать однозначно).
3. **Переадресация ввода-вывода** (позволяет переадресовывать ввод и вывод информации на различные устройства и системы; **конвойеризация — вариант перенаправления (переадресации) ввода-вывода**).
4. **Создание среды пользователя** (для каждого пользователя shell организует доступ к любой команде; интерпретатор shell запускается как фоновый процесс наравне с другими процессами ядра ОС и у каждого пользователя он создает среду, т.е. он постоянно поддерживает интерфейс пользователя с ОС и вычислительной системой; **интерпретатор shell является родителем для всех процессов, которые пользователь создает, а также проводником всех утилит и программ, реализованных в виде командной строки**).
5. **Поддержка командного языка** (с помощью командного языка можно администрировать систему, писать легко реализуемые программы, не уступающие программам на процедурном языке; синтез командного языка программирования shell и языка С позволил создать такие современные языковые средства, как Python, PHP, Perl и т.п.).

Интерпретатор shell. Функции. Интерпретация команд

Любая утилита, вводимая в командной строке (или конвойер программ) подвергается интерпретации.

Интерпретатор shell анализирует синтаксис утилиты и, если допущены ошибки при задании утилиты, то он сообщает об этом пользователю, в противном случае выполняет утилиту.

13.2 Обработка имен файлов, определенных через метасимволы

Интерпретатор shell. Функции. Обработка имен файлов, определенных через метасимволы

Интерпретатор shell допускает использование широкого набора метасимволов для сокращенной формы записи имен файлов.

- * — произвольная строка символов (набор или цепочка-символов).
- ? — произвольный символ (один).
- [] — альтернативный символ подстановки.

Метасимвол «*» определяет произвольную строку символов (любой набор или цепочка-символов). Может располагаться в произвольном месте имени файла. Например, пусть текущий каталог содержит файлы: fil1 fil2 fil3 fil4

Вывести на монитор содержимое этих файлов посредством утилиты **pr** можно следующей командой:

```
pr fil1 fil2 fil3 fil4
```

Используя метасимвол «*», эту команду можно записать по-другому следующими способами:

```
pr fil*
```

```
pr *
```

Последняя команда будет работать точно так же, как и первые две потому, что в текущем каталоге содержатся только четыре целевых файла.

Обратите внимание на терминологический нюанс, связанный с тем, что **имя файла — это одна строка символов**: если имя файла целиком задается метасимволом «*», то этот метасимвол определяет **произвольную строку символов в имени файла**, если же метасимвол используется только как часть задаваемого имени файла (например, «*.txt»), то он характеризует **набор или цепочку символов в имени файла!**

Метасимвол «?» соответствует единственному произвольному символу.

Вывести на монитор содержимое всех четырех файлов текущего каталога (см. пример выше) можно следующей утилитой:

```
cat fil?
```

Метасимвол «[]» соответствует альтернативному символу подстановки.

Вывести на монитор содержимое файлов текущего каталога посредством утилиты **pr** можно следующими способами:

```
pr fil[1 2 3 4 ]
pr fil[ 1 - 4 ]
pr fil[ 1 2-4 ]
```

13.3 Перенаправление ввода-вывода. Конвейеризация

Переадресация ввода-вывода

Если требуется, shell позволяет легко **перенаправить устройства ввода-вывода**:

- > — запись в файл, если файл не существует, то создается новый, иначе файл перезаписывается (жесткое перенаправление).
- < — ввод из файла (с самого начала) в команду.
- >> — запись в файл, если файл не существует, то создается новый, иначе файл дополняется с конца (мягкое перенаправление).
- << — ввод из файла не сначала (**Iseek** для установки позиции).

Интерпретатор shell. Функции. Переадресация вывода

Перенаправление вывода (>, >>):

ls > a.txt – вывод списка файлов текущего каталога в файл a.txt (жесткое перенаправление).

ls >> a.txt – вывод списка файлов текущего каталога в файл a.txt с добавлением информации в конец файла, если он уже существовал; если файл не существовал, то он будет создан и в него запишется результат выполнения команды **ls** (мягкое перенаправление).

`cat a1.c a2.c > a3.c` – объединение файлов a1.c и a2.c в один файл a3.c

Интерпретатор shell. Функции. Переадресация ввода

Перенаправление ввода (`<`, `<<`):

Перенаправление ввода всегда присутствует в утилитах, например, вот две эквивалентные команды подсчета с помощью утилиты `wc` количества строк, слов и символов в файле a.txt:

`wc < a.txt` эквивалентно `wc a.txt`

Другой пример — с помощью утилиты `mail` одновременно трем пользователям отправляется сообщение из одного и того же файла `let`:

`mail user1 user2 user 3 < let`

Мягкое перенаправление ввода применяется, если указатель чтения-записи файла находится не на его начале, а где-то в середине файла, и нужно осуществить перенаправление ввода именно с этого места (из середины файла).

Последовательность команд

`who > temp`

`sort < temp`

позволит получить список пользователей, работающих в настоящий момент в системе (`who`), в алфавитном порядке (`sort`). В процессе работы создается промежуточный файл `tmp`, который затем желательно удалить.

Интерпретатор shell. Функции. Переадресация ввода-вывода. Конвейеризация

Команда 1 | Команда 2

Shell также допускает **объединение нескольких команд в конвейер для их совместного последовательного выполнения**. Конвейеризация позволяет направить результат выполнения одной команды сразу на вход другой команде. В этом случае **информационная связь между командами в UNIX-подобных ОС осуществляется через межпроцессный канал** (через ОП). По своей сути, **конвейеризация — вариант перенаправления (переадресации) ввода-вывода**.

По сути, конвейер является простейшей программой на командном языке, которая позволяет, используя ряд синтаксических конструкций, объединять команды с программой, интерпретируемой, как единое целое. Например, с помощью операции конвейера «`|`» последнюю последовательность команд можно записать в следующем виде

`who | sort` (объединение 2-х утилит в конвейер — выход команды `who` поступает на вход команды `sort`, при этом никаких дополнительных файлов не создается).

Примеры:

`ls | pr -3` – список файлов в текущем каталоге, выведенный в три колонки;

`ls | wc -w` – количество файлов в текущем каталоге;

`who | wc -l` – количество пользователей, работающих в настоящий момент в системе — список пользователей, работающих в настоящий момент в системе (`who`) и количество строк в указанном файле (в данных из конвейера);

`file * | grep "C source" | cut -d : -f 1` – имена файлов текущего каталога, которые написаны на языке программирования C, полученные объединением сразу трех утилит в конвейер — проанализировать все файлы текущего каталога и вывести их характеристики (`file`), затем из этих файлов отобрать только написанные на языке программирования C или C++ (`grep`), а затем вырезать имена файлов, т.е. первое поле каждой записи до разделителя двоеточие (`cut`).

13.4 Условная последовательность выполнения команд и Поддержка командного языка программирования

Интерпретатор shell. Функции. Переадресация ввода-вывода. Условная последовательность выполнения утилит

Помимо конвейеризации, интерпретатор shell позволяет объединять несколько команд в условную последовательность выполнения команд:

- **Команда 1 && Команда 2** — вторая команда выполняется только, если успешно выполнилась первая (**конъюнкция**), т.е. если первая команда сгенерирует нулевое значение кода завершения.
- **Команда 1 || Команда 2** — вторая команда выполняется при условии неудачного завершения первой команды (**дизъюнкция**), т.е. в случае, когда первая команда сгенерирует ненулевое значение кода завершения;
- **Команда&** — запуск команды в фоновом режиме (подавление вывода в консоль всякой информации).

Рассмотрим примеры, как с помощью операций **&&** и **||** формируется условная последовательность выполнения утилит:

`gcc lab.c && ./a.out` — `./a.out` будет выполняться только если трансляция исходной программы `lab.c` прошла успешно.

`gcc lab.c -o lab.exe && ./lab.exe` — `./lab.exe` будет выполняться только если компиляция исходной программы прошла успешно.

`cd foo || echo "No such directory"` — команда `echo` выполнится только в случае, если команда `cd` не смогла перейти в директорию `foo`.

Интерпретатор shell. Функции. Создание среды пользователя

Для каждого пользователя shell организует доступ к любой команде.

Для каждого сеанса работы пользователя интерпретатор shell запускается ОС как фоновый процесс.

Основная его задача — создание среды выполнения:

- интерпретирует вводимые команды, которые пользователь вводит с клавиатуры терминала.
- сохраняет все введенные команды, созданные переменные и функции.
- является родителем для всех процессов пользователя, запущенных им из командной строки.
- поддерживает выполнение своих функций — возможность использования метасимволов,

конвейеризации, перенаправления (переадресации) ввода-вывода.

Интерпретатор shell. Функции. Поддержка командного языка программирования

Командный язык является развитым языком программирования, позволяющим реализовать многообразие различных приложений. Командный язык содержит такие синтаксические конструкции, как условный оператор, операторы циклов, оператор переключателя, конструкции сравнения и т.п.

13.5 Встроенные и внешние команды интерпретатора shell

Интерпретатор shell включает встроенные и внешние команды:

- **Встроенные** — это команды, являющиеся частью интерпретатора. Они не требуют при своем выполнении запуска нового процесса.
- **Внешние** — это команды, для выполнения которых порождается дополнительный процесс (например, `grep`, `ls`, `lp`, `more`, `sort` и т.д.).

Встроенные команды языка shell

Встроенные команды являются частью интерпретатора shell. Они не требуют при своем выполнении запуска отдельного процесса. Встроенных команд немного:

- **:** — нуль функция, всегда возвращает истинное значение, то есть 0 (в отличии от C, в shell истина 0, ложь 1).
- **break** — выходит из цикла (применяется в конструкциях циклов **for**, **while**, **case**, **until**).
- **continue** — продолжает цикл, начиная следующую итерацию (следующий шаг в операторах цикла).
- **cd** — изменение текущего каталога (*change dir*).
- **echo** — записывает свои внешние аргументы в стандартный файл вывода.
- **eval** — считывает аргумент и выполняет указанную команду с этим аргументом.
- **exec** — выполняет команду, но не в рамках shell, а в UNIX (в остальном она очень похожа на **eval**).
- **exit** — выход из интерпретатора shell (также, как и в языке C). Можно выгрузить из своего окружения и сам интерпретатор shell.
- **export** — экспортирует shell-переменные между различными программными приложениями, написанными на shell (*например, между скриптами разных пользователей*).
- **pwd** — отображает имя текущего каталога (полный путь от корневого каталога к текущему рабочему каталогу, в контексте которого будут исполняться вводимые команды).
- **read** — считывает строку текста (цепочку символов) из стандартного файла ввода (клавиатуры). Это могут быть любые символы, окончание ввода характеризуется нажатием клавиши ENTER (переводом на новую строку «\n»).
- **readonly** — преобразует shell-переменную в переменную «только для чтения» (режим **readonly**), т.е. делает из переменной константу.
- **return** — выход из shell-функции с отображением кода возврата.
- **set** — управление параметрами для стандартного файла ввода. Позволяет отображать или устанавливать переменные оболочки и среды.
- **shift** — смещает влево командную строку внешних аргументов. Позволяет осуществлять цикл по внешним аргументам.
- **test** — оценивает условное выражение.
- **times** — отображает имя пользователя и системные промежутки времени для процессов, которые выполняются с помощью shell.
- **trap** — перехват у ОС обработки сигналов (при получении сигнала выполняет команду или последовательность команд).
- **type** — интерпретирует, каким образом shell принимает имя в качестве команды.
- **ulimit** — отображает или устанавливает ресурсы shell.
- **umask** — отображает или устанавливает права доступа к файлам (изменяет права доступа, которые присваиваются новым файлам и каталогам по умолчанию).
- **unset** — удаляет из памяти shell-переменную или функцию (все переменные и функции постоянно находятся в памяти shell до тех пор, пока вы не перезагрузите эту программу).
- **wait** — ожидает окончания процесса-сына и сообщает о его завершении.

Внешние команды

Внешние команды shell — это **остальные команды, для выполнения которых порождается дополнительный процесс**. Например:

- **cmp**
- **cp**
- **file**
- **find**
- **grep**
- **lp**
- **ls**

- **more**
- **mv**
- **sort**
- **wc**
- и т.д.

Команда echo

Echo — это встроенная команда языка *shell*, существующая только внутри интерпретатора (это не системная утилита, у нее нет исполняемого файла). Она имеет только одно назначение — выводить строку текста в стандартный файл вывода (монитор или терминал), но применяется очень часто в различных скриптах, программах, и даже для редактирования конфигурационных файлов. Синтаксис команды **echo**:

echo < опции > <строка>

Опции:

- **-n** — не выводить перевод строки;
- **-e** — включить поддержку вывода Escape последовательностей;
- **-E** — отключить интерпретацию Escape последовательностей.

Это все опции, если включена опция **-e**, то вы можете использовать такие Escape последовательности для вставки специальных символов:

- **\c** — удалить перевод строки;
- **\t** — горизонтальная табуляция;
- **\v** — вертикальная табуляция;
- **\b** — удалить предыдущий символ;
- **\n** — перевод строки;
- **\r** — символ возврата каретки в начало строки.

Команда echo. Примеры

ВВОД: echo "abc\n def \nghi"

ВЫВОД:

abc\n def \nghi

ВВОД: echo -e "abc\n def \nghi"

ВЫВОД:

*abc
def
ghi*

Команда exit

Exit — это встроенная команда языка *shell*, закрывающая оболочку со статусом *N*:

exit N

Если *N* не задано, то код состояния выхода — это **код последней выполненной команды**.

Если в теле одной из функций использовать команду **exit**, то выполнение *shell*-скрипта мгновенно завершится. При использовании в сценариях *shell*, значение, указанное в качестве аргумента команды **exit**, возвращается оболочке как код выхода.

Команда exit. Пример

```
exit 0      # Команда exit закрывает оболочку со статусом 0. В соответствии с
            # соглашениями, 0 указывает на успешное завершение, в то время как
            # ненулевое значение означает ошибку.
```

14. Понятие shell-переменной. Специальные shell-переменные. Конструкции командного языка программирования expr, let, test. Примеры реализаций

14.1 Понятие shell-переменной. Специальные shell-переменные

Понятие shell-переменной

Для эффективного взаимодействия команд в рамках программ на командном языке интерпретатор разрешает использование специальных shell-переменных — обычных идентификаторов, которые объявляются и инициализируются присваиванием и, к которым при использовании добавляется символ «\$» (или «%», или «#»).

Любая shell-переменная — обычный набор (цепочка) символов, поскольку в интерпретаторе shell нет типов переменных. Объявляются и инициализируются присваиванием или вводом. При использовании добавляется «\$». Например:

```
aa = world      # Объявление shell-переменной с помощью оператора присваивания
$aa            # Использование shell-переменной aa (добавление $ означает
               # значение переменной)
read bb        # Чтение в shell-переменную $bb с терминала
```

Экранирующие и специальные символы

Кроме shell-переменных интерпретатор поддерживает специальные символы:

- `команда` или { команда } — подстановка результата выполнения команды (символ «`» находится в верхнем левом углу клавиатуры и называется «тупое» ударение).
- \ — отменяет специальный смысл непосредственно следующего за ним специального символа.
- '...' — отменяет специальный смысл всех символов, заключенных в одиночные кавычки (символы «'»). Экранируется вся строка или цепочка символов.
- "..." — отменяет специальный смысл всех символов, заключенных в двойные кавычки (символы «"»), исключая символы «\», «"», «\$», «'».

Специальные знаки и символы

- ; — разделяет несколько команд, указанных в одной строке. По умолчанию интерпретатор shell допускает только одну команду на строку.
- () — объединяет несколько команд в единое задание.
- # — комментарий.

Специальные системные shell-переменные

Помимо переменных, введенных пользователем, shell позволяет использовать специальные системные переменные:

- \$ — все переменные интерпретатора shell.
- \$# — количество внешних аргументов выполняемой программы.
- \$1, \$2... — внешний аргумент 1, 2 и т.д.
- \$? — возвращаемое значение последней выполненной программы, команды shell или инструкции.
- \$\$ — идентификатор текущего процесса.

Присваивание значений shell-переменным

1. Присваивание значений shell-переменным — это одна из конструкций интерпретатора shell.

Выполняется либо оператором присваивания, либо командой read, читающей пользовательский ввод:

```
a = < значение >      # Объявление shell-переменной с помощью оператора присваивания
read < имя >          # Чтение в shell-переменную с терминала
```

Команда `read` считывает строку текста (цепочку любых символов, включая специальные, до символа перевода строки, т.е. до тех пор, пока пользователь не нажмет на клавиатуре клавишу `<ENTER>`) из стандартного файла ввода, т.е. из терминала. Считанная информация запишется в shell-переменную с указанным именем.

Пример

```
lect141-1.sh:
a=5      # Объявление переменной и задание ей значения 5 с помощью оператора присваивания
b=hello # Объявление переменной и задание ей значения hello
read c   # Чтение в shell-переменную $c из терминала
echo a = $a $b c = "$c" # Вывод в стандартный файл вывода (монитор)
                      # внешних аргументов команды echo
```

Примечание: Если указать `"$c"`, то выведутся все считанные из терминала символы, включая повторяющиеся пробелы. Если указать `$c`, то выведутся все символы, но цепочки подряд идущих пробелов при этом будут урезаны до одного.

Сохраняем в файл `lab` и делаем файл исполняемым:

```
chmod u + x lab
```

Запускаем файл на исполнение:

```
./lab
```

Ответ: `a = 5 hello c = <введенное значение с клавиатуры>`

Пример 2

```
echo "Input your name: "
read name
echo "Your name is $name"
```

Ответ: `Your name is <введенная с клавиатуры цепочка символов>`

14.2 Вычисление арифметических выражений в shell

2. **Вычисление арифметических выражений** — это одна из конструкций интерпретатора shell.

Осуществляется двумя способами:

2.1. Командой

```
expr < арифметическое выражение >
```

Операции для `expr` (и `let`): `+, -, *, /, %` (последняя операция — взятие остатка от деления).

Возвращает значение арифметического выражения.

Для того, чтобы получить результат выполнения команды `expr`, нужно поместить ее внутри символов «тупого» ударения ````.

2.2. Командой

```
let < арифметическое выражение >
```

Операции только для `let`: `>, <, >=, <=, =, !=, &, |, !` (помимо предыдущих пяти операций, добавлены операции сравнения, последние три из которых: и `&`, или `|`, не `!`). Это позволяет использовать конструкцию `let` для рассмотрения логических выражений).

Возвращает значение арифметического выражения.

Команда `let` удобнее в применении, т.к. позволяет использовать обычные двойные кавычки.

Пример 1

lect142-1.sh:

```
a=2          # Объявление переменной $a и присвоение ей значения
a=`expr $a + 7` # Значение переменной $a увеличивается на 7
b=`expr $a / 3` # Создается переменная $b и ей присваивается значение $a / 3
c=`expr $a '*' $b` # Здесь метасимвол «*» экранируется одиночными
                     # или двойными кавычками
echo $a b = $b c = $c # Вывод в стандартный файл вывода (монитор)
```

Ответ: 9 b = 3 c = 27

Пример 2

lect142-2.sh:

```
a=1          # Объявление переменной $a и присвоение ей значения
while let "a<=3" # Цикл будет выполняться до тех пор, пока значение
                  # переменной $a <= 3
do
    echo $a Hello # Вывод в стандартный файл вывода (монитор) значения переменной
                  # и цепочки символов
    let "a=a+1"   # Увеличение значения переменной на единицу
done
```

Ответ:

1 Hello
2 Hello
3 Hello

14.3 Вычисление арифметических выражений в shell (продолжение)

2. Сравнение цепочек символов — это одна из конструкций интерпретатора shell, относящаяся к вычислению арифметических выражений. Осуществляется все той же

2.3. Командой

expr <цепочка символов> : <цепочка символов>

Возвращает количество совпадающих символов. Сравнение начинается с самого начала цепочки — в цепочках abcd и cd совпадений найдено не будет.

Пример

lect143-1.sh:

```
x=abcde      # Объявление переменной $x и присвоение ей цепочки
               # из пяти символов
y=`expr "$x" : 'abc'` # Создается переменная $y и ей присваивается результат
                      # сравнения $x с цепочкой символов abc
z=`expr "$x" : 'abd'` # Создается переменная $z и ей присваивается результат
                      # сравнения $x с цепочкой символов abd
w=`expr $x : ".*"`  # Создается переменная $w и ей присваивается результат
                      # сравнения $x с произвольной цепочкой символов ".*"
echo $y $z $w       # Вывод в стандартный файл вывода (монитор)
```

Ответ: 3 0 5

14.4 Конструкция shell test. Работа с файлами

3. Проверка файлов, числовых величин (условных выражений) и цепочек символов осуществляется командой **test** — одной из конструкций интерпретатора shell.

Проверка выражений на истинность:

test < выражение >

эквивалентно:

[выражение]

Возвращает 0, если условное выражение истинно и 1, если ложно.

3.1. Проверка файлов:

test -флаг имя_файла

Флаги test (в разных UNIX-подобных ОС они могут отличаться):

-f — существует ли файл и является ли он обычным;

-d — существует ли каталог;

-b — существует ли файл и является ли он блок-ориентированным специальным файлом;

-c — существует ли файл и является ли он байт-ориентированным специальным файлом;

-p — существует ли файл и является ли он каналом;

-S — существует ли файл и является ли он гнездом;

-s — существует ли файл с размером > 0 (непустой файл);

-r — проверка файла на чтение, если он существует;

-w — проверка файла на запись, если он существует;

-x — проверка файла на исполнение, если он существует;

и др.

Работа с файлами. Примеры:

test -f lab # существует ли обычный файл lab в текущем каталоге?

эквивалентно:

[-f lab]

Сравнение чисел

3.2. Сравнение чисел при помощи команды **test**:

test <число> отношение <число>

Поскольку в интерпретаторе shell нет типов переменных и любая shell-переменная — это обычный набор (цепочка) символов, то для сравнения переменных в качестве чисел указываются специальные отношения, взятые из языка программирования FORTRAN:

- -lt — меньше, <
- -le — меньше или равно, ≤
- -gt — больше, >
- -ge — больше или равно, ≥
- -eq — равно, ==
- -ne — не равно, !=

Если использовать знаки из правой части, то команда **test** будет сравнивать цепочки символов, а не числа.

Сравнение чисел. Пример

a=6 # Объявление переменной \$a и присвоение ей значения

test \$a -lt 10 # Переменная \$a меньше 10?

эквивалентно:

[\$a -lt 10]

echo \$? # Вывод в стандартный файл вывода (монитор) значения,

```
# возвращаемого командой test
```

Ответ: 0

14.5 Конструкция shell test. Операции с условиями. Сравнение цепочек символов

Сравнение цепочек символов

3.2. Проверка цепочки символов при помощи команды test:

test <цепочка>=<цепочка> — проверка равенства двух цепочек

test <цепочка>!=<цепочка> — проверка на различие (не равенство) двух цепочек

test -n <цепочка> — проверка существования (аналог grep) — цепочка существует и не равна null

test -z <цепочка> — проверка отсутствия цепочки символов — цепочка существует и равна null

Сравнение цепочек символов. Примеры:

Проверка равенства:

x=abc

test "\$x" = "abc"

echo \$?

Проверка на различие:

x=abc

["\$x" != "abc"]

echo \$?

Ответ: 0

Ответ: 1

Примечание: Если указать "\$x", то выводятся все считанные символы, включая повторяющиеся пробелы.

Если указать \$x, то выводятся все символы, но цепочки подряд идущих пробелов при этом будут урезаны до одного.

Пример 2:

```
if test -n "$1"          # Если задан (первый) внешний аргумент,
    echo "first arg is $1" # то его значение выводится в
                           # стандартный файл вывода (монитор).
else
    echo "first arg not exist" # В противном случае аргумент не задан,
                               # выводится сообщение об этом.
fi
```

Возможности смешанной проверки

Использование операций **!** (НЕ), **-o** (ИЛИ), **-a** (И) в команде **test** позволяет проводить смешанные проверки.

15. Конструкции командного языка программирования while, until, for, if, case, trap. Примеры реализаций

15.1 Условные операторы if, case. Примеры реализаций

Сравнение осуществляется с помощью конструкций интерпретатора if и case.

Конструкция if

```
if < условие А >
    then <список команд В>      # Если условие А выполняется
    elif < условие С >          # Если условие А не выполняется
        then <список команд D>  # Если условие А не выполняется, но выполняется
                                # условие С
    else <список команд D>    # Если оба условия не выполняются
fi                           # Конец конструкции if
```

Ключевое слово **elif** означает "else if".

Конструкция if. Пример

lect151-1.sh:

```
read a                      # Чтение в shell-переменную $a из терминала
if [ $a -gt 5 ]; then       # Если значение переменной $a больше 5, то
    echo "Привет"           # в стандартный файл вывода (монитор)
                            # выводится цепочка символов
else                        # Если значение переменной $a меньше или равно 5, то
    if [ $a -eq 5 ]          # Если значение переменной $a равно 5, то
        then
            echo "Hello"
        else
            # В противном случае значение переменной $a
            # меньше 5, тогда
            echo "Goodbuy"
    fi
fi
```

Ответ:

3
Goodbuy

5
Hello

125
Привет

Конструкция case

Сравнение с помощью конструкции case, которая определяет, в каких моделях присутствует shell-переменная или выражение.

```
case <переменная или выражение> in
    модель 1) список команд А;;
    модель 2) список команд В;;
    ....
```

```
модель Н) список команд Н;;
*) список команд по умолчанию;;
```

esac

Условие «*)» в последней модели обозначает произвольную цепочку символов. Как только значение shell-переменной или shell-выражения будет совпадать с shell-переменной или shell-выражением в одной из моделей, то сразу же будет осуществлен выход из конструкции case, **возвращаемое значение конструкции case** при этом будет равно 0.

Конструкция case. Пример

lect151-2.sh:

```
x=abc      # Объявление переменной $x и присвоение ей цепочки из трех символов
case $x in # Переменная $x используется в конструкции case
    ?*a*) echo "в цепочке символов имеется символ а";;
    a*) echo "цепочка символов начинается с символа а";;
        *) echo "в цепочке символов нет символа а";;
esac       # Конец конструкции case
```

Ответ: цепочка символов начинается с символа а

15.2 Конструкции циклов while, until. Примеры реализаций

Циклы осуществляются операторами **while** (пока условие истинное), **until** (до тех пор, пока условие ложное), **for**.

Конструкция while

```
while < истинное условие >
do
    <список команд>
done
```

Конструкция while. Пример

lect152-1.sh:

```
while : # Бесконечный цикл, т.к. нуль функция «:» всегда возвращает
        # истинное значение, то есть 0
do
    echo "введите q, чтобы завершить процесс"
    read a          # Чтение в shell-переменную $a из терминала.
    if [ "$a" = "q" ] # Если значение цепочки символов, содержащейся в
                      # переменной $a равно значению цепочки символов "q", то
        then exit 0  # команда exit закрывает оболочку со статусом 0.
    fi
done
```

Конструкция until

```
until < ложное условие >
do
    <список команд>
done
```

Конструкция until. Пример

lect152-2.sh:

```
x=0          # Объявление shell-переменной $x и присвоение ей значения
```

```

until test $x -eq 3      # Пока числовое значение переменной $x не равно трем,
do
    echo "Hello"          # в стандартный файл вывода (монитор)
    # выводится цепочка символов,
    x=`expr $x + 1`        # а числовое значение переменной $x
    # увеличивается на единицу.

done
# После того, как числовое значение переменной $x станет равным трем,
# произойдет выход из цикла until.

```

Ответ:

```

Hello
Hello
Hello

```

15.3 Конструкция цикла for. Примеры реализации

Циклы осуществляются операторами while (пока условие истинное), until (до тех пор, пока условие ложное), for.

Конструкция for

```

for <переменная> [in список ]
do
    <список команд>
done

```

Необязательное выражение в квадратных скобках нужно для того, чтобы значение переменной перебирало все варианты в указанном списке. Если список отсутствует, то shell-переменная будет перебирать значения внешних аргументов вызываемой программы.

Конструкция for. Пример 1

```

lect153-1.sh:
for i in 1 2 65 10      # Переменная $i перебирает все варианты в списке 1 2 65 10
do
    echo number $i       # В стандартный файл вывода (монитор)
    # выводится слово и значение переменной
done

```

Ответ:

```

number 1
number 2
number 65
number 10

```

Конструкция for. Пример 2

```

lect153-2.sh:
n=0                      # Объявление shell-переменной $n и присвоение ей значения
for i                     # shell-переменная $i будет перебирать значения
do
    n=`expr $n + 1`        # Числовое значение переменной $n увеличивается на единицу.
                           # Альтернативные записи: n=$(expr $n + 1) и let "n=n+1"
echo "argument $n: $i"   # В стандартный файл вывода (монитор) выводится

```

```

# цепочка символов с номером аргумента $n и значением
done          # i-го внешнего аргумента.

# После того, как цикл будет произведен по всем внешним аргументам, произойдет выход
из цикла for.

```

Ответ:

Сохраняем в файл prim.sh, затем делаем файл исполняемым:

```
chmod u +x prim.sh
```

Запуск файла с четырьмя внешними аргументами (в данном случае всего пять внешних аргументов, нулевой аргумент — само имя программы prim.sh):

```
./prim.sh one two 33 саша
```

```
argument 1: one
```

```
argument 2: two
```

```
argument 3: 33
```

```
argument 4: саша
```

15.4 Обработка сигналов в shell. Конструкция trap

С помощью встроенной команды **trap** интерпретатор shell может **перехватывать обработку сигналов** у ОС.

Конструкция trap

Команда **trap** перехватывает у ОС обработку сигналов (при получении сигнала(ов) выполняет команду или последовательность команд).

```
trap "<список команд>" <список сигналов>
```

<список команд> — команды (или функции), которые **необходимо выполнить при обработке сигналов**.

<список сигналов> — список сигналов, которые **необходимо перехватывать (кроме SIGKILL и SIGSTOP)**.

Сигнал задается либо в виде символьной константы, либо в виде кода этого сигнала (например, сигнал SIGINT имеет код 2).

Сигнал **SIGKILL** прекращает выполнение процесса. Это довольно специфический сигнал, который посыпается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнала процесса. Иногда он также посыпается ОС (например, при завершении работы системы). **Сигналы SIGSTOP и SIGKILL** предназначены **только для остановки или завершения процесса и не могут игнорироваться или перехватываться**, то есть обрабатываться при помощи определенной пользователем процедуры.

Конструкция trap. Пример

При поступлении сигнала SIGINT или SIGTSTP будут выполнены три утилиты: **ls**, **who**, **ps** lect154-1.sh:

Установка нового обработчика из трех команд при поступлении сигналов прерывания, указанных далее:

```

trap "
ls
who
ps
" SIGINT SIGTSTP
sleep 3 # Замедление выполнения работы программы на 3 секунды для того, чтобы
        # пользователь успел нажать < Ctrl+C > или < Ctrl+Z >.

```

В данном случае сигналы задаются через их символьные константы:

- SIGINT (код 2) — сигнал прерывания < Ctrl+C > с терминала, вызывающий завершение процесса;

- SIGTSTP (код 23) — сигнал остановки с терминала <Ctrl+Z>, приводящий к остановке процесса.

15.5 Понятие функции в shell. Пример реализации

В shell можно использовать **функции**. Формат для определения функции следующий:

```
имя_функции ()
{
    команда 1
    команда 2
    .....
    команда N
}
```

Множество команд в фигурных скобках образует тело функции. Функции запускаются простым **упоминанием их имен**. Именем функции может быть любая цепочка символов. Вызов функции равен вызову команды, при условии что они загружены в окружение интерпретатора команд shell. Когда вы указываете имя функции, вместо нее интерпретатор подставляет тот набор команд, который вы объединили в виде одной функции.

Функцию не следует описывать в теле другой функций. В остальных случаях описывать функцию можно в любом месте программы. Однако если вызвать функцию до ее описания, то возникнет ошибка "function: not found" («функция не найдена»).

Понятие функции в shell. Пример

Написать shell-скрипт подсчета количества итераций бесконечного цикла до тех пор, пока пользователь не прервёт его выполнение через <Ctrl+C> (сигнал прерывания SIGINT, код 2) или <Ctrl+\> (сигнал прерывания SIGQUIT, код 3).

lect155-1.sh:

```
i=0                                # Объявление shell-переменной $i (счетчика итераций)
                                    # и присвоение ей значения.

handl() {                            # Обработчик сигнала - функция handl:
    echo " Количество итераций равно $i" # В стандартный файл вывода (монитор)
                                            # выводится число итераций цикла.
    exit 0                                # Команда exit закрывает оболочку
                                            # со статусом 0.
}

trap "handl" 2 3                    # Установка нового обработчика handl
                                    # при поступлении сигналов SIGINT или SIGQUIT.

while :                             # Бесконечный цикл, т.к. нуль функция «:» всегда возвращает
do                                  # истинное значение, то есть 0:
    i=`expr $i + 1`                 # Числовое значение переменной $i (число итераций)
                                    # увеличивается на единицу.
                                    # Альтернативные записи: i=$(expr $i + 1) и let "i=i+1"
    echo $i                          # В стандартный файл вывода (монитор)
                                    # выводится значение переменной $i.
    sleep 1                           # Замедление выполнения работы программы на 1 секунду для того,
                                    # чтобы пользователь успел нажать <Ctrl+C> или <Ctrl+\>.

done
```

15.6-15.7 Примеры реализаций shell-скриптов

Пример 1

Написать shell-скрипт вывода в столбец внешних аргументов в его вызове. Если при вызове скрипта внешние аргументы отсутствуют, то вывести идентификатор текущего процесса.

```
lect156-1.sh:
if [ $# -eq 0 ]; # Если количество внешних аргументов выполняемой программы
# равно нулю:
then
    echo "идентификатор процесса равен $$"
else # Иначе у выполняемой программы есть внешние аргументы:
    echo "внешние аргументы в вызове данного скрипта:"
    while [ $1 ] # Цикл рассматривает первый внешний аргумент,
# если он не нулевой, то
do
    echo $1 # в стандартный файл вывода (монитор)
# выводится значение внешнего аргумента.
    shift # Смещение влево командной строки внешних аргументов.
# Теперь вместо первого внешнего аргумента используется
# второй внешний аргумент. Это позволяет
# осуществить цикл по внешним аргументам.
done
fi
```

Ответ:

```
./lect156-1.sh one two 123
```

внешние аргументы в вызове данного скрипта:

```
one
```

```
two
```

```
123
```

```
./lect156-1.sh
```

идентификатор процесса равен 3177

Пример 2

Написать shell-процедуру подсчета символов в цепочке символов, вводимой с клавиатуры.

```
lect156-2.sh:
```

```
echo "Введите цепочку символов:"
read a # Чтение в shell-переменную $a из терминала
echo "введенная строка символов: $a"
leng=`expr "$a" : '.*'` # Создается переменная $leng и ей присваивается результат
# сравнения $a с произвольной цепочкой символов ".*" - т.е.
# количество совпадающих символов (длина цепочки $a).
# Примечание: Здесь указано "$a", т.е. учитываются все считанные из терминала
# символы, включая повторяющиеся пробелы.
echo "цепочка символов состоит из $leng символов"
```

Ответ:

Введите цепочку символов:

Hello ALL!

введенная строка символов: Hello ALL!

цепочка символов состоит из 10 символов

Пример 3

Написать shell-скрипт подсчета количества файлов в текущем каталоге, которые содержат текстовую информацию, или в имени файла содержат слово text. Список таких файлов направьте в файл text.txt.

Во-первых, нужно определить, какие файлы в текущем каталоге — текстовые. Для этого подойдут две утилиты, уже упоминавшиеся ранее:

- **file** (1-ый вариант) — определяет тип файла и анализирует его содержимое. Она анализирует первые записи файла и на основе этого анализа выводит свою характеристику указанного файла. Она может показать тип файла — текстовый (в какой кодировке и на каком языке программирования он написан) или двоичный (исполняемый файл или картинка и т.д.).
- **find** (2-ой вариант, короткое решение) — поиск файла в любых каталогах ФС UNIX с указанными параметрами. Имеет неограниченное количество внешних аргументов, т.е. характеристики поиска можно задавать очень большим количеством внешних аргументов.

Пример 3 (1-ый вариант)

lect156-3v1.sh:

```
if [ -f text.txt ]; then # Если в текущем каталоге целевой файл уже существует,
# то для того, чтобы он не содержал лишней информации, он удаляется:
    unlink text.txt # Удаление жесткой ссылки на файл или самого файла,
# - последняя жесткая ссылка на файл.

fi
ubuntu@ubuntu:~/test$ ls
a.out b.txt jj.sh jmp.c kk new.sh null one.sh read text.txt
a=`ls` # shell-переменной $a присваивается результат выполнения утилиты ls.
# Теперь $a содержит имена файлов текущего каталога, разделенные пробелами.
n=0      # Объявление shell-переменной $n и присвоение ей значения (счетчик файлов,
# которые содержат текстовую информацию).

for i in $a; do # Цикл для каждого имени файла. Переменная $i перебирает все имена
# файлов из списка в переменной $a:
```

```
ubuntu@ubuntu:~/test$ file *
a.out: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-arm
hf.so.3, BuildID[sha1]=2b79870a176b35e8adde971386c7cdd239fabe73, for GNU/Linux 3.2.0, not stripped
b.txt: ASCII text
jj.sh: POSIX shell script, UTF-8 Unicode text executable
jmp.c: C source, UTF-8 Unicode text
kk: empty
new.sh: Bourne-Again shell script, UTF-8 Unicode text executable
null: ASCII text
one.sh: UTF-8 Unicode text
read: empty
text.txt: ASCII text
```

t=`file \$i` # Переменной \$t присваивается результат выполнения утилиты file,
которая в файле с именем \$i анализирует первые 10 записей и выводит имя файла,
затем символ двоеточия, затем тип этого файла и другую информацию.

```
ubuntu@ubuntu:~/test$ file * | grep text
b.txt: ASCII text
jj.sh: POSIX shell script, UTF-8 Unicode text executable
jmp.c: C source, UTF-8 Unicode text
new.sh: Bourne-Again shell script, UTF-8 Unicode text executable
null: ASCII text
one.sh: UTF-8 Unicode text
text.txt: ASCII text
```

echo \$t | grep text >/dev/null # результат команды file из переменной \$t
передается через межпроцессный канал по конвейеру на вход утилиты grep,
которая выводит только те записи, которые содержат шаблон "text".

```
# >/dev/null - вывод grep будет перенаправляться на нулевое устройство
# ввода-вывода для того, чтобы он не отображался на экране.

if [ $? -eq 0 ]; then # Если возвращаемое значение последней команды (grep)
    # равняется нулю, то шаблон "text" содержится в записи.
    # Примечание: Если grep возвращает единицу, то шаблон в записи не содержится,
    # а возвращаемое значение 2 обозначает ошибку.
    echo $i >>text.txt # Имя файла $i, содержащего текстовую информацию, или
    # содержащего слово text в своем имени, дописывается (мягкое перенаправление)
    # в файл text.txt.
    n=`expr $n + 1` # Счетчик найденных файлов увеличивается.

fi
done
# Выводится результат подсчета:
echo количество файлов текущего каталога, содержащих текстовую информацию $n
```

Ответ:

```
ubuntu@ubuntu:~/test$ cat jj.sh
if [ -f text.txt ]
then unlink text.txt
fi
a=`ls`
n=0
for i in $a
do
t=`file $i`
echo $t | grep text > /dev/null
if [ $? -eq 0 ]
then
    echo $i >> text.txt
    n=`expr $n + 1`
fi
done
echo количество файлов текущего каталога содержащих текстовую информации $n

ubuntu@ubuntu:~/test$ ./jj.sh
количество файлов текущего каталога содержащих текстовую информации 6
```

Пример 3 (2-ой вариант)

lect156-3v2.sh:

```
find . -exec file {} \; | grep text | cut -d : -f 1 | tee text.txt | wc -l
: '
```

Утилита find предусматривает неограниченное количество аргументов и применяется для поиска в текущем каталоге всех файлов:

. (каталог поиска) - это отправной каталог (текущий каталог), с которого find начинает поиск файлов по всем подкаталогам, находящимся внутри;
опция -exec используется для выполнения произвольных команд для найденных файлов.
В данном случае, выполнить команду file для получения подробной информации о каждом файле.

Затем полученная информация передаются через межпроцессный канал (с помощью конвейеризации) на вход утилиты grep, которая выводит только те записи, которые содержат шаблон "text".

Затем полученный набор записей, содержащих искомый шаблон передается по конвейеру на вход утилиты cut, которая вырезает имена файлов, т.е. первое поле каждой записи до символа-разделителя двоеточие ":".

Затем полученный набор записей, содержащих имена файлов, по конвейеру передается на вход утилите tee, которая записывает вывод любой команды в один или несколько файлов.

В данном случае утилита `tee` осуществляет два вывода одной и той же информации:

1. В файл `text.txt` сохраняется набор записей с именами искомых файлов;
2. По конвейеру на вход утилиты `wc -l`, подсчитывающей количество строк из файла стандартного ввода (полученного через конвейер). В результате количество записей с именами файлов будет подсчитано как количество строк.

Ответ:

```
ubuntu@ubuntu:~/test$ find . -exec file {} \; | grep text | cut -d : -f 1 | tee text.txt | wc -l
7
```

Пример 4

Написать shell-скрипт сортировки обычных файлов из текущего каталога по размеру.

lect156-4.sh:

```
find . -type f -exec ls -l {} \; | sort +4n | more
:
```

Утилита `find` предусматривает неограниченное количество аргументов и применяется для поиска в текущем каталоге всех обычных файлов:

. (каталог поиска) - это отправной каталог (текущий каталог), с которого `find` начинает поиск файлов по всем подкаталогам, находящимся внутри;
параметр `-type f` - искать только обычные файлы.

опция `-exec` используется для выполнения произвольных команд для найденных файлов.

В данном случае, выполнить утилиту `ls` с флагом `-l`, выводящую имя файла с подробной информацией о нем - владелец, группа, права доступа, время последнего обновления, размер в байтах и др. Результат выполнения программы на данном этапе представлен на картинке ниже:

```
ubuntu@ubuntu:~/test$ find . -type f -exec ls -l {} \;
-rw-rw-r-- 1 ubuntu ubuntu 0 Apr  7 17:42 ./kk
-rwxrwxr-x 1 ubuntu ubuntu 8504 Apr  7 20:13 ./a.out
-rw-rw-r-- 1 ubuntu ubuntu 87 Apr  9 21:19 ./null
-rw-rw-r-- 1 ubuntu ubuntu 0 Apr  7 18:43 ./read
-rw-rw-r-- 1 ubuntu ubuntu 651 Apr  7 20:13 ./jmp.c
-rwxrwxrwx 1 ubuntu ubuntu 229 Apr 12 18:25 ./j2.sh
-rwxrwxrwx 1 ubuntu ubuntu 270 Apr 12 18:20 ./j1.sh
-rwxrwxrwx 1 ubuntu ubuntu 588 Apr  7 19:22 ./one.sh
-rwxrwxrwx 1 ubuntu ubuntu 265 Apr 12 17:39 ./new.sh
-rwxrwxrwx 1 ubuntu ubuntu 320 Apr 12 18:13 ./jj.sh
-rwxrwxrwx 1 ubuntu ubuntu 274 Apr  7 18:06 ./b.txt
-rw-rw-r-- 1 ubuntu ubuntu 43 Apr 12 18:20 ./text.txt
```

После того, как для каждого обычного файла в текущем каталоге будет применена утилита `ls`, полученная информация передается через межпроцессный канал (с помощью конвейеризации) на вход утилиты `sort`, которая выводит текстовые строки в отсортированном порядке. Опция `+4n` используется для сортировки по числовому значению четвертого поля (размеру файла в байтах).

Затем информация, отсортированная по размеру файлов, передается по конвейеру на вход Утилиты `more`, которая осуществляет постраничный вывод содержимого стандартного файла ввода (полученного через конвейер) на монитор.

Вместо утилиты `more` можно использовать также утилиты `cat` (вывод содержимое обычного файла на монитор), `head` (вывод первых строк файла), `tail` (вывод последних строк файла) и т.д.

Ответ:

```
ubuntu@ubuntu:~/test$ find . -type f -exec ls -l {} \; | sort +4n | more
-rw-rw-r-- 1 ubuntu ubuntu 0 Apr  7 17:42 ./kk
-rw-rw-r-- 1 ubuntu ubuntu 0 Apr  7 18:43 ./read
-rw-rw-r-- 1 ubuntu ubuntu 43 Apr 12 18:20 ./text.txt
-rw-rw-r-- 1 ubuntu ubuntu 87 Apr  9 21:19 ./null
-rwxrwxrwx 1 ubuntu ubuntu 229 Apr 12 18:25 ./j2.sh
-rwxrwxrwx 1 ubuntu ubuntu 265 Apr 12 17:39 ./new.sh
-rwxrwxrwx 1 ubuntu ubuntu 270 Apr 12 18:20 ./j1.sh
-rwxrwxrwx 1 ubuntu ubuntu 274 Apr  7 18:06 ./b.txt
-rwxrwxrwx 1 ubuntu ubuntu 320 Apr 12 18:13 ./jj.sh
-rwxrwxrwx 1 ubuntu ubuntu 588 Apr  7 19:22 ./one.sh
-rw-rw-r-- 1 ubuntu ubuntu 651 Apr  7 20:13 ./jmp.c
-rwxrwxr-x 1 ubuntu ubuntu 8504 Apr  7 20:13 ./a.out
```

Пример 5

Написать программу вывода имен всех объектных файлов каталога /home/ubuntu/test, превышающих по объему число, указанное в качестве первого внешнего аргумента скрипта.

lect156-5.sh:

```
dir=`pwd` # Запоминаем текущий каталог.
:
shell-переменной $dir присваивается результат выполнения команды pwd.
Теперь $dir содержит имя текущего каталога (полный путь от корневого
каталога к текущему рабочему каталогу).
```

```
cd /home/ubuntu/test # Переход в целевой каталог
files=`file * | grep object | cut -d : -f 1`
: 'Утилита
file *
```

в каждом из файлов текущего каталога анализирует первые 10 записей и выводит имя файла, затем символ двоеточия, затем тип этого файла и другую информацию.

```
ubuntu@ubuntu:~/test$ file *
a.out: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=2b79870a176b35e8adde971386c7cd239fabe73, for GNU/Linux 3.2.0, not stripped
b.txt: ASCII text
file1.exe: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=ca771d57718008ee6cd33de51a8171650bf237a, for GNU/Linux 3.2.0, not stripped
file2.exe: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=74fe37fa8033601c1a906702b69db47a1a1ccfe9, for GNU/Linux 3.2.0, not stripped
file3.exe: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=ca771d57718008ee6cd33de51a8171650bf237a, for GNU/Linux 3.2.0, not stripped
file4.exe: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=276992e425538566d1c674ee77feb021d6843617, for GNU/Linux 3.2.0, not stripped
j1.sh: UTF-8 Unicode text
j2.sh: UTF-8 Unicode text
j3.sh: ASCII text
jj.sh: UTF-8 Unicode text
jmp.c: C source, UTF-8 Unicode text
kk: empty
main.o: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=f2d0c80d33ca940a49544404d0c09c90c69dc614, for GNU/Linux 3.2.0, not stripped
new.sh: Bourne-Again shell script, UTF-8 Unicode text executable
null: ASCII text
one.sh: UTF-8 Unicode text
read: empty
test1: directory
text.txt: ASCII text
```

Организован конвейер: результат команды file передается через межпроцессный канал по конвейеру на вход утилиты grep, которая выводит только те записи, которые содержат шаблон "object":

| grep object

```
ubuntu@ubuntu:~/test$ file * | grep object
a.out: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=2b79870a176b35e8adde971386c7cd239fabe73, for GNU/Linux 3.2.0, not stripped
file1.exe: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=ca771d57718008ee6cd33de51a8171650bf237a, for GNU/Linux 3.2.0, not stripped
file2.exe: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=74fe37fa8033601c1a906702b69db47a1a1ccfe9, for GNU/Linux 3.2.0, not stripped
file3.exe: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=ca771d57718008ee6cd33de51a8171650bf237a, for GNU/Linux 3.2.0, not stripped
file4.exe: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=276992e425538566d1c674ee77feb021d6843617, for GNU/Linux 3.2.0, not stripped
main.o: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
f.so.3, BuildID[sha1]=f2d0c80d33ca940a49544404d0c09c90c69dc614, for GNU/Linux 3.2.0, not stripped
```

Затем полученный набор записей, содержащих искомый шаблон, передается по конвейеру

на вход утилиты `cut`, которая вырезает первый столбец до символа разделения ":"
| `cut -d : -f 1`

```
ubuntu@ubuntu:~/test$ file * | grep object | cut -d : -f 1
a.out
file1.exe
file2.exe
file3.exe
file4.exe
main.o
```

Результат выполнения конвейера из трех утилит, содержащий имена всех объектных файлов текущего (целевого) каталога, сохраняется в shell-переменную `$files`.

```
'  
for i in $files # Цикл по переменной $files, значение переменной $i будет  
do          # принимать имена файлов.  
n=`du -b $i | awk '{print $1}'` # В переменную $n записывается размер  
                                # указанного файла $i в байтах.  
: ' Утилита du с опцией -b выводит размер файла в байтах и имя файла,  
содержащееся в shell-переменной $i.
```

```
ubuntu@ubuntu:~/test$ du -b a.out  
8504  a.out
```

Результат работы утилиты `du` передается через межпроцессный канал по конвейеру на вход утилиты `awk`, которая вырезает первое поле, содержащее размер файла:

```
| awk '{print $1}'
```

Одинарные кавычки применяются для того, чтобы экранировать знак доллара, чтобы shell не восприняла последовательность символов "\$1" как первый внешний аргумент. Вместо утилиты `awk` можно использовать также утилиту `cut`, редактор `sed` и т.д.

```
'  
if [ $n -gt $1 ]; then # Если значение переменной $n больше значения первого  
                      # внешнего аргумента $1, то в стандартный файл вывода  
echo $i             # (монитор) выводится имя этого файла.  
fi  
done  
cd $dir # Возврат в каталог, в котором мы находились до выполнения процедуры.
```

Ответ:

```
ubuntu@ubuntu:~/test$ ubuntu@ubuntu:~/test$ cat j3.sh  
dir=`pwd`  
cd /home/ubuntu/test  
files= file * | grep object | cut -d : -f 1  
for i in $files  
do  
n=`du -b $i | awk '{print $1}'`  
if [ $n -gt $1 ]  
then echo $i  
fi  
done  
cd $dir  
  
ubuntu@ubuntu:~/test$ ./j3.sh 8148  
a.out  
file2.exe  
file4.exe  
main.o
```

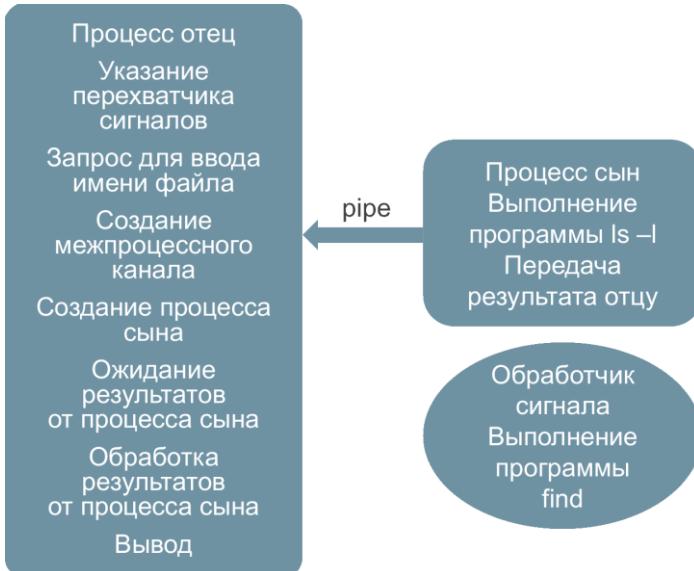
16 Примеры реализаций многозадачных программных приложений в shell

16.1-16.2 Пример 1. Реализация многозадачного приложения

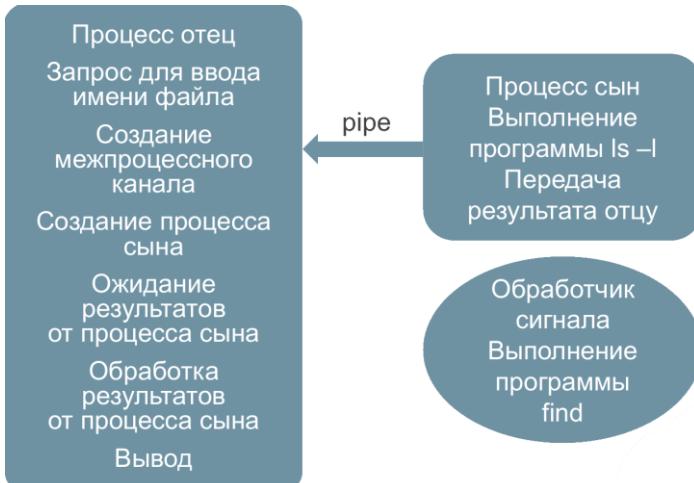
Пример 1. Постановка задачи

Написать программу определения типа файла, указанного при запросе. Предусмотреть обработку сигнала прерывания от клавиатуры. При поступлении 3-х сигналов вывести все файлы текущего каталога, написанные на языке программирования С.

Подходы к реализации



Подходы к реализации (основная программа)



lect16_example_1.sh:

```

# Обработчик сигнала - функция handle:
i=0 # Счетчик сигналов прерывания
handle() {
    i=`expr $i + 1` # Счетчик сигналов прерывания увеличивается.
    # Альтернативные записи:
    # i=$(expr $i + 1) и let "i=i+1"
    if test $i -eq 3; then # При поступлении 3-х сигналов (сравниваются
        # числовые выражения)
        file * | grep "C source" | cut -d : -f 1 # Вместо "C source" может быть
  
```

```
# "C program" - в зависимости от ОС
```

```
: ' Утилита
```

```
file *
```

в каждом из файлов текущего каталога анализирует первые 10 записей и выводит имя файла, затем символ двоеточия, затем тип этого файла и другую информацию.

Организован конвейер: результат команды file передается через межпроцессный канал по конвейеру на вход утилиты grep, которая выводит только те записи, которые содержат шаблон "C source":

```
| grep "C source"
```

Затем полученный набор записей, содержащих искомый шаблон передается по конвейеру на вход утилиты cut, которая вырезает первый столбец до символа разделения ":"

```
| cut -d : -f 1
```

Результат выполнения конвейера из трех утилит выведет имена всех файлов текущего каталога, написанных на языке программирования С.

```
'
```

```
fi
```

```
if [ $flag -eq 0 ]; then # Если ввод еще не произведен, то запрос повторяется:
```

```
    echo -e "введите имя файла:"
```

```
fi
```

```
}
```

```
:
```

Установка нового обработчика hand1 при поступлении сигнала прерывания SIGINT (код 2), по умолчанию вызывающего завершение процесса при нажатии < Ctrl+C > в терминале.

```
'
```

```
trap "hand1" SIGINT
```

```
# Основная программа:
```

```
flag=0          # Переменная для фиксации факта ввода пользователя
```

```
echo -e "введите имя файла:" # Опция "-e" включает поддержку вывода
```

```
                      # Escape последовательностей
```

```
read name        # Чтение в переменную name имени файла,
```

```
                      # введенного с терминала
```

```
flag=1          # Ввод произведен
```

```
# Если во время ввода поступит сигнал, то значение флага не изменится.
```

```
# Если ввод завершится успешно, то флаг станет равным единице.
```

```
:
```

Утилита

```
ls -lda
```

выводит имена всех файлов в текущем каталоге, включая скрытые файлы, имена которых начинаются с точки. Рассмотрим, за что отвечает каждый флаг по отдельности:

-l - выводить список с подробной информацией о файлах - владелец, группа, права доступа, время последнего обновления, размер и др.;

-d - выводить только информацию о директории, без ее содержимого (подавляет вывод общего размера

всех файлов в каждом подкаталоге), полезно при рекурсивном выводе;

-a - отображать все файлы, включая скрытые, имена которых начинаются с точки.

Добавление переменной \$name с именем файла выведет информацию только об этом файле или каталоге.

Организован конвейер: результат команды ls передается через межпроцессный канал по конвейеру на вход утилиты cut, которая вырезает первый байт из полученной записи:
| cut -c 1-1

В итоге, shell-переменная b будет содержать результат выполнения конвейера из двух утилит ls и cut - один символ, характеризующий тип файла \$name.

```
b=`ls -lda $name | cut -c 1-1`
```

```
:
```

Определение типа файла исходя из символа, содержащегося в shell-переменной \$b. Как только значение shell-переменной или shell-выражения (case <переменная или выражение>) будет совпадать с shell-переменной или shell-выражением в одной из моделей, то сразу же будет осуществляться выход из конструкции case:

```
case $b in
  '-' ) echo "файл $name - обычный" ;;
  'd' ) echo "файл $name - директория" ;;
  'p' ) echo "файл $name - канал" ;;
  'c' ) echo "файл $name - специальный байт-ориентированный файл" ;;
  'b' ) echo "файл $name - специальный блок-ориентированный файл" ;;
  's' ) echo "файл $name - сокет (гнездо)" ;;
  'l' ) echo "файл $name - символьная ссылка" ;;
  *) default ;;
esac
```

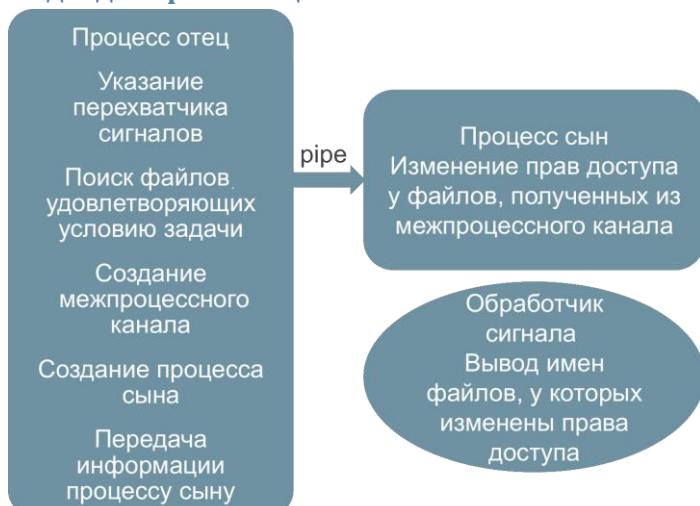
16.3-16.4 Пример 2. Реализация многозадачного приложения

Пример 2

Написать программу поиска в текущем каталоге файлов, созданных за последнюю неделю, размер которых превышает 55 байт. Полученную информацию переслать через межпроцессный канал в параллельный процесс, где изменить права доступа выделенных файлов на введенные с клавиатуры. Предусмотреть возможность неоднократного прерывания по сигналу SIGINT.

При каждом прерывании осуществлять вывод имен тех файлов, у которых изменены права доступа.

Подходы к реализации



```

lect16_example_2.sh
# Обработчик сигнала - функция prer:
prer() {
    if test -s a.txt; then # Если файл a.txt существует в текущей директории и его
                           # размер больше нуля (он содержит нужную информацию),
        cat a.txt          # то выводятся имена файлов (содержимое этого файла),
                           # у которых изменены права доступа
    else                  # Если файл a.txt пустой или он не существует
        echo "\nПрава доступа пока не менялись"
    fi
    if [ $flag -eq 0 ]; then # Если ввод еще не произведен, то запрос повторяется:
        echo "Введите права доступа для файлов"
    fi
}
:
Установка нового обработчика prer при поступлении сигнала прерывания SIGINT (код 2),
по умолчанию вызывающего завершение процесса при нажатии < Ctrl+C > в терминале.
'
trap "prer" SIGINT

# Основная программа:
flag=0 # Переменная для фиксации факта ввода пользователя
:
В файл a.txt будут записаны имена файлов, права доступа у которых изменены.
Поскольку будет использоваться мягкое перенаправление (дописывание) в файл a.txt,
то в случае, если он уже существует, дописывание в него может дать неверный
результат на выходе, т.е. он будет включать посторонние данные.
'
if test -s a.txt; then # Поэтому, если файл a.txt уже существует
                       # в текущей директории и его размер больше нуля,
    rm a.txt           # то он удаляется
fi
echo "Введите права доступа для файлов"
read b # Чтение в переменную $b прав доступа, введенных с терминала
flag=1 # Ввод произведен
# Если во время ввода поступит сигнал, то значение флага не изменится.
# Если ввод завершится успешно, то флаг станет равным единице.
:
Утилита find предусматривает неограниченное количество аргументов и применяется
для поиска в текущем каталоге всех файлов размером больше 55 байт, которые были
изменены за последнюю неделю:
. (каталог поиска)-это отправной каталог (текущий каталог), с которого find начинает
Поиск файлов по всем подкаталогам, находящимся внутри.
параметр -type f - искать только обычные файлы.
критерий -size + - искать файлы больше заданного размера (c - размер задан
в байтах).
критерий -mtime - - поиск по времени модификации файла, заданным в количестве
прошедших суток;
знак минус и цифра семь определяют время, меньшее или равное аргументу (найти файлы,
которые обновлялись в течение последних семи дней).

```

Примечание: Результат, аналогичный полученному командой `find` можно получить с помощью комбинации утилит `du` и `ls`, организовав конвейер.

Результат работы утилиты `find` через межпроцессный канал (с помощью конвейеризации) передается утилите `tr`, которая меняет все символы перевода строки на пробелы.

Эта замена производится потому, что утилита `find` каждое имя файла выводит в отдельной строке, а чтение всей информации будет единожды производиться с помощью конструкции `read`, которая считывает до перевода на новую строку.

Затем имена файлов, разделенные пробелами, передаются через межпроцессный канал (с помощью конвейеризации).

Стоит отметить, что данный подход не позволит корректно обрабатывать файлы, в именах которых содержатся символы пробела, т.к. каждое слово в именах таких файлов будет обрабатываться как отдельный файл.

```
'  
find . -type f -size +55c -mtime -7 | tr "\n" " " | (  
    # Код процесса-сына:  
    read a          # Считываение всех данных из межпроцессного канала  
                  # в переменную $a  
    for i in $a; do # Цикл по переменной $a, значение переменной $i будет принимать  
                  # имена файлов  
        sleep 2      # Замедление выполнения работы программы на 2 секунды для того,  
                  # чтобы пользователь успел нажать < Ctrl+C >  
        chmod $b $i # К файлам с именем в переменной $i права доступа изменяются  
                  # на права, находящиеся в переменной $b  
    # При успешном выполнении любая утилита возвращает ноль, поэтому с помощью  
    # конструкции test проверяется возвращаемое значение последней команды $?  
    if test $? -eq 0; then # Если возвращаемое значение последней команды  
                          # равняется нулю,  
        echo $i >>a.txt    # то имя файла $i дописывается (мягкое  
                          # перенаправление) в файл a.txt  
    else # В противном случае сообщается, что утилиты chmod не выполнилась  
          # (пользователь не является владельцем этого файла и не может изменить  
          # права доступа к нему)  
        echo "Некорректные права доступа"  
    fi  
done # Выход из цикла после изменения прав доступа у файла  
)
```