



# UNIX

Поскольку процесс в системе UNIX может выполняться в двух режимах, режиме ядра или режиме задачи, он пользуется в каждом из этих режимов отдельным стеком. Стек задачи содержит аргументы, локальные переменные и другую информацию относительно функций, выполняемых в режиме задачи. Когда процесс выполняет специальную инструкцию, он переключается в режим ядра, выполняет операции ядра и использует стек ядра.

Стек ядра содержит записи активации для функций, выполняющихся в режиме ядра. Элементы функций и данных в стеке ядра соответствуют функциям и данным, относящимся к ядру, но не к программе пользователя, тем не менее, конструкция стека ядра подобна конструкции стека задачи. Стек ядра для процесса пуст, если процесс выполняется в режиме задачи.

В качестве другого примера, поясняющего ту ключевую роль, которую играет иерархия процессов, давайте рассмотрим, как UNIX инициализирует саму себя при запуске сразу же после начальной загрузки компьютера. В загрузочном образе присутствует специальный процесс, называемый *init*. В начале своей работы он считывает файл, сообщающий о количестве терминалов. Затем он разветвляется, порождая по одному процессу на каждый терминал. Эти процессы ждут, пока кто-нибудь не зарегистрируется в системе. Если регистрация проходит успешно, процесс регистрации порождает оболочку для приема команд. Эти команды могут породить другие процессы и т. д. Таким образом, все процессы во всей системе принадлежат единому дереву, в корне которого находится процесс *init*.

## Ядро процесса

## Суть процесса

Ядро представляет собой специальную программу,, которая постоянно находится в оперативной памяти и работает, пока работает операционная система. Ядро управляет всеми таблицами, используемыми для отслеживания процессов и других ресурсов.

Загружается в память во время начальной загрузки и немедленно запускает необходимые процессы, в частности процесс инициализации операционной системы - init

По умолчанию, этот процесс исполняет программу /sbin/init. При загрузке ядра можно указать, что запускать в качестве init. Это может быть полезно при восстановлении системы после аварии, но при нормальной работе не используется.

полнит регистры информацией, ранее сохранённой в объекте. Программисты, пишущие части ядра операционной системы, часто говорят, что этот объект ядра и есть процесс; иначе говоря, процесс — это структура данных, создаваемая в ядре операционной системы для поддержки выполнения пользовательской программы. Конечно, это тоже никоим образом не определение, это скорее отражение определённой точки зрения.



Системные процессы являются частью ядра и всегда расположены в оперативной памяти.

Системные процессы не имеют соответствующих им программ в виде исполняемых файлов и запускаются особым образом при инициализации ядра системы. Выполняемые инструкции и данные этих процессов находятся в ядре системы, таким образом, они могут вызывать функции и обращаться к данным, недоступным для остальных процессов

К системным процессам можно отнести и процесс начальной инициализации, init. Хотя init не является частью ядра, и его запуск происходит из выполняемого файла

### 1) Идентификация (id процесса, родителя, сеанса, группы процессов)

### 2) Разграничение полномочий (id пользователя, группы; эффективный id)

### 3) Память: у процесса состоит из нескольких сегментов

- Сегмент кода- бинарный исполняемый код, хранится программа в виде машинного кода, выполняющаяся в данном процесса

Сегмент текста содержит команды, полученные из кода, написанного пользователем

в данном процессе. Изменять содержимое этой области памяти процесс не может, что позволяет при запуске нескольких экземпляров одной и той же программы держать в памяти только одну копию её кода. В секции данных

- Сегмент данных - содержит статические и внешние переменные. Пременные с классами памяти extern и static

◦ Сегмент данных и кода инициализируется информацией из файла a.out, когда программа загружается в память

- Сегмент BSS - секция неинициализированных статических и внешних переменных, с классами памяти extern, static

• Сегмент стека - используется для сохранения активационных записей, содержащих переменные среды, позиционные переменные, локальные переменные, аргументы вызовов функций, значения регистров, возвращаемые значения функций и другую информацию. Код, создающий и уничтожающий эти записи, генерируется компилятором и вставляется в начало и конец каждой функции языка С. При каждом вызове функции запись активации размещается в стеке. При возвращении из функции запись активации освобождается. Место расположения стека и направление его роста машинно-зависимы. Попытка обращения к памяти между окончанием сегмента данных и вершиной стека приводит к ошибке обращения к памяти.

- Сегмент кучи - то, откуда берет память malloc

- Динамические сегменты

- Могут быть другие сегменты, это зависит от конкретной системы

#### **4) Состояние регистров центрального процессора (включая счетчик команд, регистр флагов, указатель стека и все регистры общего назначения)**

Когда процесс находится вне состояния выполнения, содержимое регистров ЦП для этого процесса хранится в специальной структуре данных в ядре. Когда процесс продолжает выполнение - данные из структуры копируются обратно в регистры

Регистр	Назначение
PC	Программный счетчик - указывает на текущую строку кода.
PS	Указывает состояние процессора.
SP	Указывает на вершину стека.
FP	Указывает на текущий фрейм стека.

#### **5) Готовность и приоритет процесса (эти характеристики интересны подсистеме ядра - планировщику времени ЦП)**

#### **6) Аргументы командной строки, переменные окружения**

Обычно новые запускаемые программы наследуют окружение от тех, кто их запустил - это позволяет передавать определенные настройки системы всем имеющимся процессам, с возможностью запуска некоторых процессов с измененными настройками

Наследование окружение обусловлено не свойством ОС, а библиотечными функциями, через которые обычно запускаются новые программы. С точки зрения ядра - окружение для каждой программы задается свое

Функция `int setenv(const char* name, const char* value, int overwrite)` - устанавливает новое значение переменной:

- Если переменной не было - значение устанавливается в любом случае
- Если была - новое значение устанавливается при `overwrite = 0`, т.е. `overwrite` запрещает менять ПО

#### **7) Текущий и корневой каталоги**

#### **8) Счетчики потребленных ресурсов (процессорного времени, памяти и тд)**

#### **9) Таблица файловых дескрипторов**

Каждый процесс UNIX имеет **контекст**, под которым понимается вся информация, требуемая для описания процесса. Эта информация сохраняется, когда выполнение процесса приостанавливается, и восстанавливается, когда планировщик предоставляет процессу вычислительные ресурсы.

**Контекст процесса в ОС UNIX состоит из нескольких частей:**

##### **• Адресное пространство процесса в пользовательском режиме**

Сюда входят код, данные и стек процесса, а также другие области, например, разделяемая память или код и данные динамических библиотек.

##### **• Управляющая информация**

Ядро использует две основные структуры для управления процессом - `proc` и `user`. Сюда же входят данные, необходимые для отображения виртуального адресного пространства процесса в физическую память.

##### **• Среда процесса**

Переменные среды процесса, значения которых задаются в командном интерпретаторе или в самом процессе с помощью системных вызовов, а также наследуются порожденным процессом от родительского и обычно хранятся в нижней части стека. Среду процесса можно получать или изменять с помощью функций

##### **• Аппаратный контекст**

Сюда входят значения общих и ряда системных регистров процессора, в частности, указатель текущей инструкции и указатель стека

## **Контекст процесса**

## **Виртуальное адресное пр - во**

## Стек в ВАП

ная идея виртуальной памяти, напомним, состоит в том, что исполнительные адреса, фигурирующие в машинных командах, считаются не адресами физических ячеек памяти, а некими абстрактными *виртуальными адресами*. Всё множество виртуальных адресов называется *виртуальным адресным пространством*. Виртуальные адреса преобразуются центральным процессором — а точнее, специальной схемой в составе процессора, которая называется MMU, *memory management unit* — в адреса ячеек памяти (*физические адреса*) по некоторым правилам, причём эти правила могут динамически изменяться. В современных условиях преобразование виртуальных адресов интересует только два её базовых свойства: каждая задача в системе имеет своё собственное (виртуальное) адресное пространство, причём одна и та же физическая памяти может при необходимости соответствовать как разным адресам в виртуальном пространстве одной задачи, так и адресам из разных пространств.

Виртуальная память реализуется и автоматически поддерживается ядром ОС UNIX.

Максимальная память адресного пр-ва ограничен длиной вирт. адреса и на 32-битных машинах составляет 4Гб

Контекст ядра - структуры адресного пространства

## User Area

## Пользовательская область

Размер стека определяется при старте программы (обычно 8Мб) - речь идет об области виртуальных адресов

Физическая память: система изначально выделяет под стек столько памяти, чтобы там можно было разместить переменные окружения, слова командной строки и массив указателей на них. По надобности ядро выделяет дополнительные страницы физической памяти

При этом рост стека подчиняется двум ограничениям (при выходе за них- процесс уничтожится):

- На размер самого стека
- На общий объем виртуальной памяти, выделенных данному процессу

## Терминалы и процессы демоны

Область кучи и отображения mmap ограничены только объемом адресного пространства, соответственно растут навстречу друг другу, постепенно занимая его

ОС сохраняет информацию о процессах в структурах данных, размещаемых в памяти ядра: в дескрипторах процесса и пользовательских областях.

*Пользовательская область* — это системный сегмент данных небольшого фиксированного размера, который содержит информацию, необходимую при исполнении этого процесса, например дескрипторы открытых файлов, реакцию на сигналы, информация о системных ресурсах.

Для получения/установки информации, хранящейся в user area, использовать системные вызовы.

Кроме атрибутов процесса, пользовательская область содержит стек, которым ядро пользуется при выполнении системных вызовов. Функции ядра не могут размещать свои записи активации на пользовательском стеке, ведь тогда другая нить пользовательского процесса могла бы подменить параметры функций или адрес возврата и вмешаться в работу кода ядра.

Если в рамках процесса исполняется несколько нитей, то пользовательская область должна содержать соответствующее количество стеков, чтобы каждая нить могла выполнять системные вызовы независимо от других.

\*Пользовательская область не является непосредственно доступной для пользователя, тк находится в памяти ядра. Даже если ядро отображено в адресное пространство процесса, на соответствующих страницах памяти стоят атрибуты, делающие эти страницы доступными только в системном режиме.

ступает серверная программа (так называемый **демон**), предоставляющая доступ — **sshd**, **telnetd** или их аналог.

Отметим заодно, что при предоставлении удалённого доступа к машине также используется программная эмуляция терминала; в роли эмулятора вы-

## Последовательная работа в UNIX:

**Ядро представляет собой специальную программу, которая постоянно находится в оперативной памяти и работает, пока работает операционная система.** Ядро управляет всеми таблицами, используемыми для отслеживания процессов и других ресурсов.

Изначально при запуске системы во время начальной загрузки, ядро загружается в оперативную память и немедленно запускает необходимые процессы. Порождает первый процесс `init` (`pid = 1`), являющийся прародителем всех пользовательских процессов, запуская программу `/sbin/init`. При загрузке ядра можно указать, что запускать в качестве `init`. Он читает командный файл `/etc/inittab` (см. [inittab\(4\)](#)) и запускает все остальные задачи в системе, используя `fork(2)` и `exec(2)`.

Изначально существует только процесс. Затем вызовом `fork()` создается новый процесс - копия родителя.

**Дочерний процесс наследует** все отображённые на память файлы и вообще все сегменты адресного пространства, все открытые файлы, идентификаторы группы процессов и сессии, реальный и эффективный идентификаторы пользователя и группы, ограничения `rlimit`, текущий каталог, а также ряд других параметров.

**Дочерний процесс НЕ наследует:** идентификатор процесса, идентификатор родительского процесса, а также захваченные участки файлов.

Очень часто сразу же после вызова `fork()` следует вызов функции из семейства `exec()` - отличаются они переданными параметрами, но выполняют одну и ту же функцию - полное замещение текущего процесса новой программой, переданной в качестве аргумента. В UNIX это единственный вариант запустить новую программу.

**Одно из свойств процесса** - Состояние регистров центрального процессора (включая счетчик команд, регистр флагов, указатель стека и все регистры общего назначения)

Когда процесс находится вне состояния выполнения, содержимое регистров ЦП для этого процесса хранится в специальной структуре данных в ядре. Когда процесс продолжает выполнение - данные из структуры копируются обратно в регистры

Регистр	Назначение
<b>PC</b>	Программный счетчик - указывает на текущую строку кода.
<b>PS</b>	Указывает состояние процессора.
<b>SP</b>	Указывает на вершину стека.
<b>FP</b>	Указывает на текущий фрейм стека.

## Что касается памяти:

Ядро располагается в оперативной памяти ОС. Когда программа запускается на выполнение, ядро выделяет ей место в оперативной памяти, при этом совпадение виртуальных адресов, сгенерированных компилятором, с физическими адресами необязательно.

ОС сохраняет информацию о процессах в структурах данных, размещаемых в памяти ядра: в дескрипторах процесса и пользовательских областях.

**Пользовательская область** — это системный сегмент данных небольшого фиксированного размера, который содержит информацию, необходимую при исполнении этого процесса, например дескрипторы открытых файлов, реакцию на сигналы, информация о системных ресурсах.

Кроме атрибутов процесса, пользовательская область содержит **стек ядра, которым ядро пользуется при выполнении системных вызовов**. Стек ядра содержит записи активации для функций, выполняющихся в режиме ядра. Элементы функций и данных в стеке ядра соответствуют функциям и данным, относящимся к ядру, но не к программе пользователя, тем не менее, конструкция стека ядра подобна конструкции стека задачи. Стек ядра для процесса пуст, если процесс выполняется в режиме задачи.

В процессе компиляции программа-компилятор генерирует последовательность адресов (для виртуальной машины), являющихся адресами переменных, информационных структур, инструкций и функций.

Диапазон адресов, который доступен процессу, **называется виртуальным адресным пространством**. Поскольку каждый процесс имеет свою таблицу трансляции виртуальных адресов в физические, говорят, что процесс имеет своё адресное пространство.

Теоретически, максимальный объем адресного пространства ограничен длиной виртуального адреса и на 32-битных машинах составляет 4Гб.

## Организация переменных окружения:

## **Организация переменных окружения:**

Глобальная переменная `environ` указывает на массив строк, называемый **окружением** (массив указателей на строки) Эти строки, которые заканчиваются нулевым символом, и есть **переменные окружения**, представленные в виде “имя=значение”.

В самом начале: переменные окружения лежат в конфигурационных файлах(`/etc` - для хранения системных параметров, настроек служб и приложений), потом передаются в процесс `init` и у него они хранятся на дне стека, `init` их передает (либо вызовом `exec` меняет) в дочерние процессы

Список переменных окружения — это массив указателей на строки в формате `name=value`, и эти строки обычно хранятся в верхней части адресного пространства процесса — над стеком. - Стивенс, с.269

Переменные окружения изначально лежат в пользовательском стеке процесса на его дне, затем сверху добавляются новые переменные. Вызов `putenv` (**указатель всегда должен указывать на статические данные**) использует `malloc`: ⇒ память выделяется на куче ⇒ какая-то часть переменных может лежать

1. На дне пользовательского стека
2. На куче

## **▼ ПАМЯТЬ ПРОЦЕССА**

### **3) Память: у процесса состоит из нескольких сегментов**

- *Сегмент кода*- бинарный исполняемый код, хранится программа в виде машинного кода, выполняющаяся в данном процессе

Сегмент текста содержит команды, полученные из кода, написанного пользователем

в данном процессе. Изменять содержимое этой области памяти процесс не может, что позволяет при запуске нескольких экземпляров одной и той же программы держать в памяти только одну копию её кода. В секции данных

- *Сегмент данных* - содержит статические и внешние переменные. Пременные с классами памяти `extern` и `static`
  - Сегмент данных и кода инициализируется информацией из файла `a.out`, когда программа загружается в память
- *Сегмент BSS* - секция неинициализированных статических и внешних переменных, с классами памяти `extern`, `static`
- *Сегмент стека* - используется для сохранения активационных записей, содержащих переменные среды, позиционные переменные, локальные переменные, аргументы вызовов функций, значения регистров, возвращаемые значения функций и другую информацию. Код, создающий и уничтожающий эти записи, генерируется компилятором и вставляется в начало и конец каждой функции языка С. При каждом вызове функции запись активации размещается в стеке. При возвращении из функции запись активации освобождается. Место расположения стека и направление его роста машинно-зависимы. Попытка обращения к памяти между окончанием сегмента данных и вершиной стека приводит к ошибке обращения к памяти.
- *Сегмент кучи* - то, откуда берет память `malloc`
- *Динамические сегменты*
- *Могут быть другие сегменты, это зависит от конкретной системы*

## **▼ СОЗДАНИЕ ДИНАМИЧЕСКИХ СЕГМЕНТОВ**

### **▼ ФАЙЛОВАЯ СИСТЕМА**

**Файл** - (абстрактно) некий интерфейс для работы с внешними устройствами:

Состояние файла - структура данных в ядре - дескриптор файла

**Файл** - совокупность данных, доступ к которой осуществляется по имени.

**Каталог, директория** - таблица преобразований имен файлов в адреса.

**Файловая система** - совокупность каталогов и других метаданных (т.е. системных структур данных), отслеживающих расположение файлов и свободное дисковое пространство.

## Что такое файл?

Файл представляет собой последовательность байтов, никак не организованных операционной системой. Прикладной программе файл представляется непрерывной последовательностью байтов (это не означает, что соответствующие байты занимают непрерывное пространство на физическом устройстве). Не существует разницы между файлами, представляющими двоичные данные и текстовые данные. Ваша программа несет ответственность за преобразование (если оно необходимо) внешнего формата представления данных в требуемое представление. Например, функция atoi(3) удобна для преобразования чисел из текстового вида во внутреннее машинное двоичное представление.

Каждый байт регулярного файла адресуется индивидуально. Вновь созданный файл не содержит пространства для данных. Пространство под данные предоставляется при записи в дисковый файл. Система выделяет блоки физического диска под данные по мере необходимости.

Размер файла хранится в структуре данных, называемой **inode**. В самом файле не присутствует признака конца файла.

Операционная система UNIX рассматривает файл, как универсальный интерфейс с физическим устройством. Многие системные вызовы, применимые к файлам, могут быть применены и к байт- или блок-ориентированным устройствам. Однако некоторые вызовы, наподобие lseek(2), которые изменяют позицию ввода/вывода, неприменимы к файлам некоторых устройств, например, терминальных устройств.

В пространстве процесса располагаются **сегменты**, но ядро оперирует понятием **“отображение”**, тк многие участки виртуального пространства процесса образуются с помощью механизма, реализованного сист вызовом **mmap**.

Изначально **mmap** был предназначен для отображения в виртуальную память дисковых файлов, но постепенно стал основным инструментом для работы с ВП как таковой. Во многих системах **mmap** - практически единственный способ потребовать от системы больше памяти. Также в Linux есть вызов **brk** для выделения памяти под кучу

### ▼ TEXT, DATA, BSS

**text** и **data** создаются как отображения, те при подготовке программы к запуску ядро отображает в новое ВАП фрагменты исполняемого файла, используя **mmap**.

используя механизм **mmap**. Дальнейшее во многом зависит от настроек линкера, использованных при создании этого файла; так, если в секции **.data** остаётся незадействованной часть последней (или единственной) страницы и туда помещается **.bss**, линкер может объединить эти две секции, и т. д.,

но в общем случае под **.bss** создаётся отдельное отображение, не связанное с файлом (анонимное). Также в виде анонимного отображения создаётся сегмент стека, а остальные области виртуальной памяти возникают уже во время работы процесса по его требованию.

### ▼ ОГРАНИЧЕНИЯ

Размер **стека** определяется на старте программы (лимитом RLIMIT\_STACK) около 8МБ. Рост стека подчиняется: размеру самого стека + общему объему областей виртуальной памяти, выделенных данному процессу.

**Область кучи и отображений mmap** ограничены только объемом адресного пространства и, возможно, предельным объемом памяти, выделенной одному процессу.

Областей 2 - они растут навстречу друг другу, пока не кончится лимит на количество выделенных процессу страниц.

## ▼ ММАР

- ★ **Файловый дескриптор** используется для идентификации файла при обращении к другим системным вызовам.  
Файловый дескриптор представляет собой небольшое неотрицательное целое число

- ★ **Диспетчер памяти** — это устройство, осуществляющее трансляцию виртуальных адресов в физические. Обычно диспетчер памяти осуществляет трансляцию на основе таблиц, формируемых операционной системой.

- ★ **Системный вызов lseek(2)** устанавливает позицию чтения/записи в открытом файле. Последующие вызовы read(2) и write(2) приведут к операции с данными, расположенными по новой позиции чтения/записи.

- ★ **Счетчик команд** - специальный регистр ЦП, содержит адрес ячейки памяти со следующей выбираемой командой

В **компьютерных операционных системах** пейджинг по **запросу** (в отличие от **упреждающего** пейджинга) — это метод управления **виртуальной памятью**. В системе, использующей подкачку по запросу, операционная система копирует **страницу** диска в физическую память только в том случае, если делается попытка доступа к ней и эта страница еще не находится в памяти (т. е. если происходит **ошибка страницы**). Из этого следует, что **процесс** начинает выполнение, когда ни одна из его страниц не находится в физической памяти, и будет происходить много отказов страниц до тех пор, пока большая часть **рабочего набора** страниц процесса не будет размещена в физической памяти. Это пример **ленивой загрузки** техника.

Модель **рабочего набора** утверждает, что процесс может находиться в **ОЗУ** тогда и только тогда, когда все страницы, которые он использует в настоящее время (часто аппроксимируются последними использованными страницами), могут находиться в ОЗУ. Модель представляет собой модель «все или ничего», то есть, если количество страниц, которые необходимо использовать, увеличивается, а в оперативной памяти нет места, процесс выгружается из памяти, чтобы освободить память для использования другими процессами.

## ▼ СТРАНИЧНАЯ МОДЕЛЬ

Идея виртуальной памяти - исполнительные адреса, фигурирующие в машинных командах, считаются не адресами физических ячеек памяти, а некими абстрактными виртуальными адресами. **Виртуальные адреса** преобразовываются в адреса ячеек памяти центральным процессором - а именно специальной схемой в составе процессора - MMU, по некоторым правилам, которые могут динамически изменяться.

**Виртуальные адреса** - мн-во из

Преобразование виртуальных адресов в физические практически всегда **происходит в соответствии со страницной моделью**(8.3.5). У нее есть 2 главных свойства: каждая задача в системе имеет собственное ВАП, причем одна и та же физ память может при необходимости соответствовать разным адресам в ВП одной задачи, так и адресам из разных пространств

## ▼ ММАР

Системный вызов **mmap(2)** можно использовать для установления отображения между адресным пространством процесса и файлом или запоминающим периферийным устройством. Это позволяет получать доступ к содержимому файла или устройства как к массиву байт в адресном пространстве процесса.

Преобразование виртуальных адресов в физические практически происходит в соответствии со страничной моделью. Блоки фиксированного размера, составляющие виртуальное адресное пространство, называются **страницами**. В оперативной памяти соответствующие блоки называются **страничными блоками**. Перемещаются из памяти на диск и обратно целые страницы.

ММАР реализует **подкачу по требованию**, поскольку содержимое файла не считывается напрямую с диска и изначально вообще не использует физическую оперативную память. Фактическое чтение с диска выполняется «ленивым» способом после доступа к определенному месту.

**Подкачу по требованию** - метод подкачки, при котором страницы по одной загружаются в оперативную память, только когда процесс явно обращается к ним.

Современные версии Unix позволяют отображать ресурсы хранения данных (файлы или устройства) в адресное пространство процесса. Такое отображение осуществляется при помощи **диспетчера памяти (отображает виртуальные адреса в физические)**

Система устанавливает в дескрипторах всех страниц отображённой памяти **бит отсутствия**. При первом обращении к такой странице, диспетчер памяти генерирует исключение отсутствия страницы. Ядро системы перехватывает это исключение, считывает страницу из файла или с устройства, снимает бит отсутствия в дескрипторе страницы и возвращает управление программе. Для пользовательской программы это выглядит так, как будто прочитанные с устройства данные всегда находились в странице, на которую они отображены. **Таким образом можно отображать на память не только регулярные файлы, но и устройства, поддерживающие функцию `lseek(2)`.**

Для отображения файла на память, файл должен по-прежнему открываться вызовом `open(2)` и закрываться вызовом `close(2)`.

**После отображения функцией `mmap(2)`, к содержимому файла можно обращаться, как к оперативной памяти.**

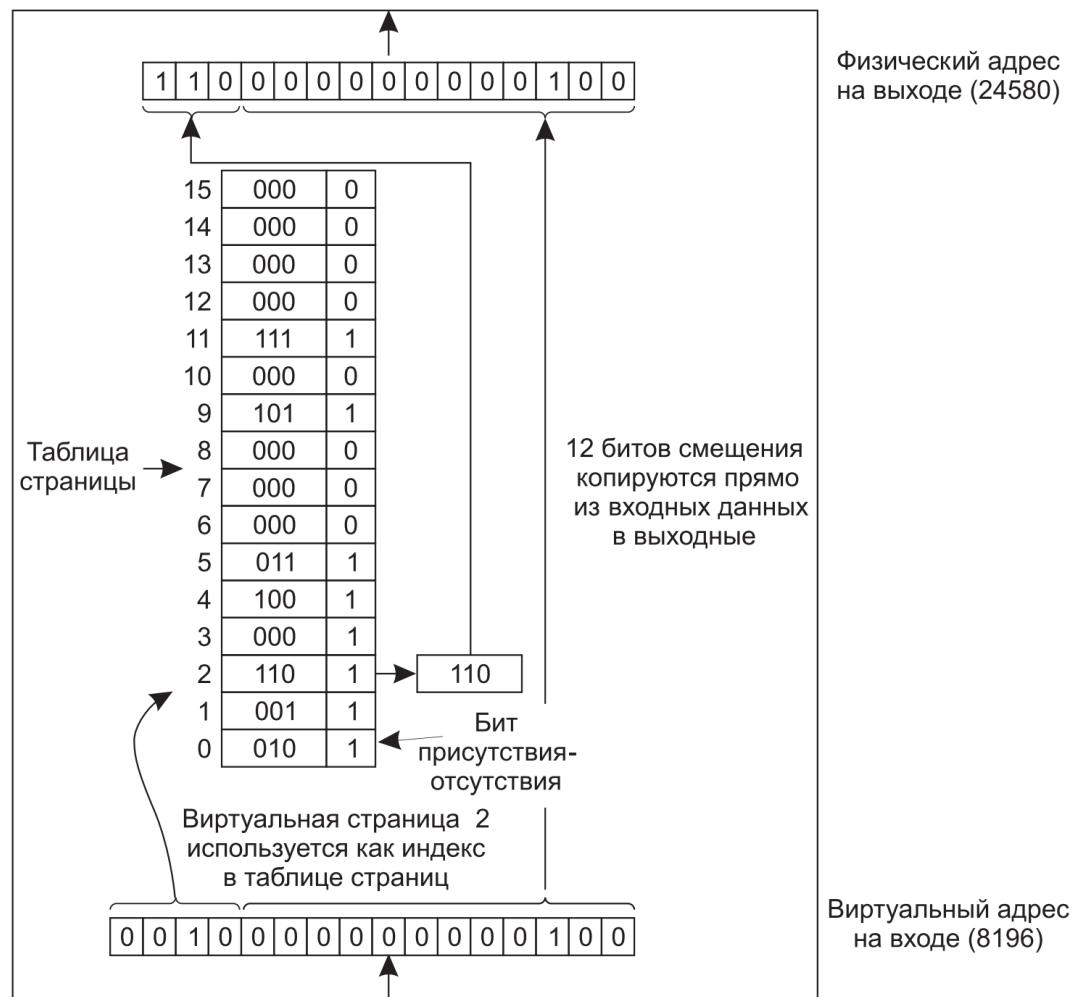
Нужный номер страницы ищется через таблицу страниц процесса по виртуальному адресу

**Прерывания и исключения** - программные решения для взаимодействия с аппаратными.

При возникновении ошибки отсутствия страницы операционная система должна выбрать выселяемую (удаляемую из памяти) страницу, чтобы освободить место для загружаемой страницы. Если предназначена для удаления страница за время своего нахождения в памяти претерпела изменения, она должна быть переписана на диске, чтобы привести дисковую копию в актуальное состояние. Но если страница не изменялась (например, она содержала текст программы), дисковая копия не утратила своей актуальности и перезапись не требуется. Тогда считываемая страница просто пишется поверх выселяемой.

Теперь рассмотрим внутреннее устройство диспетчера памяти, чтобы понять, как он работает и почему мы выбрали размер страницы, кратный степени числа 2. На рис. 3.10 показан пример виртуального адреса 8196 (001000000000100 в двоичной записи), отображенного с использованием карты диспетчера памяти с рис. 3.9. Входящий 16-разрядный виртуальный адрес делится на 4-битный номер страницы и 12-битное смещение. Выделяя 4 бита под номер страницы, мы можем иметь 16 страниц, а с 12 битами под смещение можем адресовать все 4096 байт внутри страницы.

Номер страницы используется в качестве индекса внутри **таблицы страниц** для получения номера страничного блока, соответствующего виртуальной странице. Если бит



## ▼ ВВЕДЕНИЕ В ОТОБРАЖЕНИЕ ФАЙЛОВ В АП ПРОЦЕССА

Современные версии Unix позволяют отображать ресурсы хранения данных (файлы или устройства) в адресное пространство процесса. Такое отображение осуществляется при помощи **диспетчера памяти** (**отображает виртуальные адреса в физические**)

Система устанавливает в дескрипторах всех страниц отображённой памяти **бит отсутствия**. При первом обращении к такой странице, диспетчер памяти генерирует исключение отсутствия страницы. Ядро системы перехватывает это исключение, считывает страницу из файла или с устройства, снимает бит отсутствия в дескрипторе страницы и возвращает управление программе. Для пользовательской программы это выглядит так, как будто прочитанные с устройства данные всегда находились в странице, на которую они отображены. **Таким образом можно отображать на память не только регулярные файлы, но и устройства, поддерживающие функцию lseek(2).**

Для отображения файла на память, файл должен по-прежнему открываться вызовом open(2) и закрываться вызовом close(2).

**После отображения функцией mmap(2), к содержимому файла можно обращаться, как к оперативной памяти.**

в действительности отображение происходит страницами. Начало отображаемого участка файла должно быть кратно размеру страницы (или, что то же самое, выровнено на размер страницы). Длина отображаемого участка может быть не кратна размеру страницы, но mmap(2) округляет его вверх до значения, кратного этому размеру.

Размер страницы зависит от типа диспетчера памяти, а у некоторых диспетчеров также от настроек, определяемых ядром системы. Размер страницы или, точнее, то, что данная версия Unix в данном случае считает размером страницы, можно определить системным вызовом getpagesize(2) или вызовом sysconf(2) с параметром \_SC\_PAGESIZE.

## ▼ ИСКЛЮЧЕНИЯ

**ММАР** реализует **подкачуку по требованию**, поскольку содержимое файла не считывается напрямую с диска и изначально вообще не использует физическую оперативную память. Фактическое чтение с диска выполняется «ленивым» способом после доступа к определенному месту.

**Подкачка по требованию** - метод подкачки, при котором страницы по одной загружаются в оперативную память, только когда процесс явно обращается к ним.

В [компьютерных операционных системах](#) пейджинг по **запросу** (в отличие от [упреждающего пейджинга](#)) — это метод управления [виртуальной памятью](#). В системе, использующей подкачуку по запросу, операционная система копирует [страницу](#) диска в физическую память только в том случае, если делается попытка доступа к ней и эта страница еще не находится в памяти (т. е. если происходит [ошибка страницы](#)). Из этого следует, что **процесс** начинает выполнение, когда ни одна из его страниц не находится в физической памяти, и будет происходить много отказов страниц до тех пор, пока большая часть [рабочего набора](#) страниц процесса не будет размещена в физической памяти. Это пример [ленивой загрузки](#) техника.

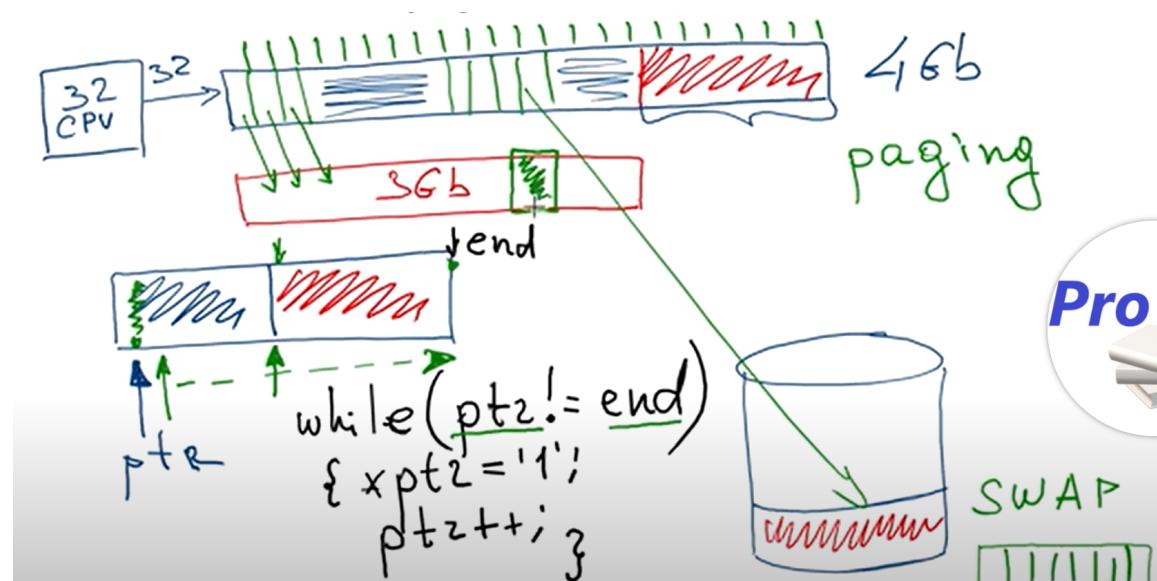
Модель **рабочего набора** утверждает, что процесс может находиться в [ОЗУ](#) тогда и только тогда, когда все страницы, которые он использует в настоящее время (часто аппроксимируются последними использованными страницами), могут находиться в ОЗУ. Модель представляет собой модель «все или ничего», то есть, если количество страниц, которые необходимо использовать, увеличивается, а в оперативной памяти нет места, процесс выгружается из памяти, чтобы освободить память для использования другими процессами.

**Прерывания и исключения** - программные решения для взаимодействия с аппаратными.

Блоки фиксированного размера, составляющие виртуальное адресное пространство, называются **страницами**. В оперативной памяти соответствующие блоки называются **страничными блоками**. Перемещаются из памяти на диск и обратно целые страницы.

Отслеживание присутствия конкретных страниц в памяти системы происходит с помощью **бита присутствия-отсутствия**. При обращении к несуществующей странице диспетчер памяти отмечает, что вызываемая страница не отображена в физическую память и **ЦП вызывает системное прерывание**, которое называется **ошибкой отсутствия страницы**. Операционная система в таком случае выбирает и сбрасывает на диск(инфу с блока) редко используемый страничный блок. После чего страница, обращение к которой вызвало ошибку отсутствия, извлекается и переносится в освободившийся страничный блок. Далее операционная система обновляет таблицы и повторно запускает команду, которая вызвала прерывание:

1. Генерируется специальное прерывание → ОС передает управление обработчику страничного прерываний (отвечает за размещение страниц в памяти)
2. Обработчик определит, какая страница нужна процессу → найдет эту страницу среди страниц хранящихся на диске → найдет то место в физической памяти, куда эта страница может быть загружена(не востребованная)



памяти. Обычно отображение страниц на физические кадры осуществляется через *таблицу страниц*, принадлежащую активной задаче (рис. 8.7). Для каждой страницы таблица содержит запись, состоящую из номера кадра и служебных атрибутов. В число атрибутов страницы обычно входит так называемый *признак присутствия*, означающий, находится ли данная страница в настоящее время в оперативной памяти или нет. При попытке обращения к странице, для которой признак присутствия сброшен, процессор инициирует исключение, называемое *страничным*; получив управление, операционная система производит, если возможно, подкачку соответствующей страницы с диска.

В UNIX System V Release 4 используется алгоритм перемещения виртуальных страниц процесса в физическую память по запросу. Обычно при запуске процесса в физическую память помещается только небольшая часть страниц, необходимая для старта процесса, а остальные страницы загружаются при страничных сбоях. Очевидно, что начальный период работы любого процесса порождает повышенную нагрузку на систему. Если при поиске виртуального адреса в соответствующем дескрипторе обнаруживается признак отсутствия этой страницы в физической памяти, то происходит страничное прерывание, и ядро перемещает эту страницу с диска в физическую память. Для поиска страницы на диске используется информация из структуры *s\_data* сегмента - либо *vnode* и *offset*, если страница типа *vnode*, либо информация о расположении анонимной страницы в области волинга с помощью информации о ее расположении там по карте *amp*.

Если в физической памяти недостаточно места для размещения затребованной процессом страницы, то ОС выгружает некоторые страницы на диск. Этот процесс осуществляется специальным процессом ядра, "выталкивателем страниц", имеющим в UNIX System V Release 4 имя *pageout*. Для принятия решения о том, какую виртуальную страницу нужно переместить на диск, процессу *pageout* нужно иметь информацию о текущем состоянии физической памяти.

## ▼ ИСПОЛЬЗОВАНИЕ ММАР

Создать файл с новыми переменными, переприсвоить значения переменных окружения

### Отображение файла на память

Системный вызов *mmap(2)* можно использовать для установления отображения между адресным пространством процесса и файлом или запоминающим периферийным устройством. Это позволяет получать доступ к содержимому файла или устройства как к массиву байт в адресном пространстве процесса.

Для отображения не требуется и не следует предварительно выделять память функцией *malloc(3C)* или каким-либо другим способом. Вызов *mmap(2)* сам выделяет необходимое виртуальное адресное пространство. В действительности, функция *malloc(3C)*, возможно, сама использует *mmap(2)* для того, чтобы запросить память у операционной системы.

Поскольку память не может быть отображена одновременно на два разных файла, не следует пытаться отображать файлы на память, выделенную через *malloc(3C)*.

`mmap(2)` возвращает начальный адрес отображённой области памяти в пределах адресного пространства вашего процесса. Далее этой памятью можно манипулировать, как любой другой памятью. `mmap(2)` позволяет процессу отобразить в память весь файл или его часть. Хотя `mmap` позволяет задавать начало и длину отображаемого участка с точностью до байта, в действительности отображение происходит страницами. Начало отображаемого участка файла должно быть кратно размеру страницы (или, что то же самое, выровнено на размер страницы). Длина отображаемого участка может быть не кратна размеру страницы, но `mmap(2)` округляет его вверх до значения, кратного этому размеру.

Размер страницы зависит от типа диспетчера памяти, а у некоторых диспетчеров также от настроек, определяемых ядром системы. Размер страницы или, точнее, то, что данная версия Unix в данном случае считает размером страницы, можно определить системным вызовом `getpagesize(2)` или вызовом `sysconf(2)` с параметром `_SC_PAGESIZE`.

### Удаление отображения страниц памяти

Системный вызов `munmap(2)` удаляет отображение страниц в диапазоне `[addr, addr+len-1]`. Последующее использование этих страниц выразится в посылке процессу сигнала `SIGSEGV`. Границы освобождаемого сегмента не обязаны совпадать с границами ранее отображеного сегмента, но надо иметь в виду, что `munmap(2)` выравнивает границы освобождаемого сегмента на границы страниц.

Также, неявное удаление отображения для всех сегментов памяти процесса происходит при завершении процесса и при вызове `exec(2)`.

Основное различие между распределённой памятью System V (`shmem`) и вводом-выводом с отображением памяти (`mmap`) состоит в том, что распределённая память System V постоянна: не будучи явно удалены, данные будут храниться в памяти и оставаться доступными до тех пор, пока система не будет отключена.

**Память `mmap`** не является постоянной между запусками прикладных программ (только если отображение не зарезервировано в файле) — сегмент памяти, созданный `mmap`, автоматически удаляется ядром системы, когда завершатся все использующие его программы-приложения.

## Разделяемая память

Разделяемая память System V IPC позволяет двум или более процессам разделять память и, следовательно, находящиеся в ней данные. Это достигается помещением в виртуальное адресное пространство процессов одной и той же физической памяти.

Того же эффекта можно достичь, отобразив в память при помощи `mmap(2)` доступный на запись файл в режиме `MAP_SHARED`. Как и в случае `mmap(2)`, разделяемые сегменты не обязательно будут отображены на одни и те же адреса в разных процессах.

Главным практическим отличием разделяемой памяти System V IPC от `mmap(2)` является то, что для `mmap(2)` нужен файл, а память System V IPC ни к какому файлу не привязана. Кроме того, использование `mmap(2)` с флагом `MAP_SHARED` приводит к тому, что система синхронизует содержимое памяти с диском, что может снизить общую производительность системы. Если вам нужно сохранить содержимое разделяемой памяти после завершения работы всех процессов приложения, `mmap(2)` оказывается удобнее, но на практике такая потребность возникает довольно редко. Поэтому разделяемая память System V IPC до сих пор широко применяется многими приложениями.

### ▼ АРГУМЕНТЫ ММАР

В ОС Unix предусмотрена возможность отображения содержимого некоторого файла в виртуальное адресное пространство процесса. В результате такого отображения появляется возможность работы с данными в файле как с обычными переменными в оперативной памяти, то есть, например, с помощью присваиваний. Отображение осуществляется системным вызовом `mmap`:

```
void *mmap(void *start, int length, int protection,
           int flags, int fd, int offset);
```

Перед обращением к `mmap` файл должен быть открыт с помощью `open`; вызов принимает дескриптор файла, подлежащего отображению, через параметр `fd`. Параметры `offset` и `length` задают соответственно позицию начала отображаемого участка в файле и его длину. Здесь нужно заметить, что и длина, и позиция должны быть кратны некоторому предопределённому числу, называемому *размером страницы*. Его можно узнать с помощью функции

```
int getpagesize();
```

## Параметры `mmap(2)`

`addr` используется для указания рекомендуемого адреса, по которому будет размещено отображение. Каким образом система располагает окончательный адрес отображения (`pa`) вблизи от `addr`, зависит от реализации. Нулевое значение `addr` дает системе полную свободу в выборе `pa`. В рамках нашего курса мы не изучаем сведений, необходимых для выбора `addr`, поэтому рекомендуется использовать нулевое значение.

`len` Длина отображаемого участка в байтах. Отображение будет размещено в диапазоне  $[pa, pa+len-1]$ . `mmap(2)` выделяет память страницами. То есть, при запросе отображения части страницы, будет отображена вся страница, покрывающая указанные байты.

`prot` Параметр `prot` определяет права доступа на чтение, запись, исполнение или их комбинацию с помощью побитового ИЛИ для отображаемых страниц. Соответствующие символьные константы определены в `<sys/mman.h>`:

`PROT_READ` страницу можно читать

`PROT_WRITE` страницу можно изменять

`PROT_EXEC` страницу можно исполнять.

`mprotect(2)` можно использовать для изменения прав доступа к отображаемой памяти

`flags` Символьные константы для этого параметра определены в `<sys/mman.h>`:

`MAP_SHARED` Если определен этот флаг, запись в память вызовет изменение отображенного объекта. Иными словами, если процесс изменяет память, отображенную с флагом `MAP_SHARED`, эти изменения будут сохранены в файле и доступны остальным процессам. Чтобы отобразить файл с `PROT_WRITE` в режиме `MAP_SHARED`, файл должен быть открыт на запись.

`MAP_PRIVATE` При указании этого флага, первое изменение отображенного объекта вызовет создание отдельной копии объекта и переназначит запись в эту копию. До первой операции записи эта копия не создается. Все изменения объекта, отображенного с флагом `MAP_PRIVATE`, производятся не над самим объектом, а над его копией. Измененные данные не сохраняются в файл, поэтому отображение файла с `PROT_WRITE` в режиме `MAP_PRIVATE` не требует ни открытия файла на запись, ни права записи в этот файл.

Либо MAP\_SHARED, либо MAP\_PRIVATE, но не оба, должны быть указаны.

MAP\_ANON Отображение «анонимной» памяти, не привязанной ни к какому файлу. В соответствии с mmap(2), это эквивалентно отображению /dev/zero без флага MAP\_ANON.

fd Файловый дескриптор отображаемого файла/устройства или -1 в сочетании с MAP\_ANON.

off Отступ от начала файла, с которого начинается отображение.

## ПРИМЕР

```
int fd, pgs;
char *p;
int size = 4096;
pgs = getpagesize();
size = ((size-1) / pgs + 1) * pgs;
/* минимальное целое число, большее либо равное
   исходному и при этом кратное размеру страницы */
fd = open("file.dat", O_RDWR);
if(fd == -1) {
    /* ... обработка ошибки ... */
}
p = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
if(p == MAP_FAILED) {
    /* ... обработка ошибки ... */
}
```

## ▼ ЛИНКЕР И СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ

\* В момент компиляции мы не знаем все статические переменные, узнаем их во время линковки

Линкер манипулирует строковыми константами, размеры которых известны во время компиляции, он не может манипулировать сегментами, размеры которых неизвестны

Тк линкер формирует сегмент данных, следовательно в нем не может лежать environment, тк знает размера environment

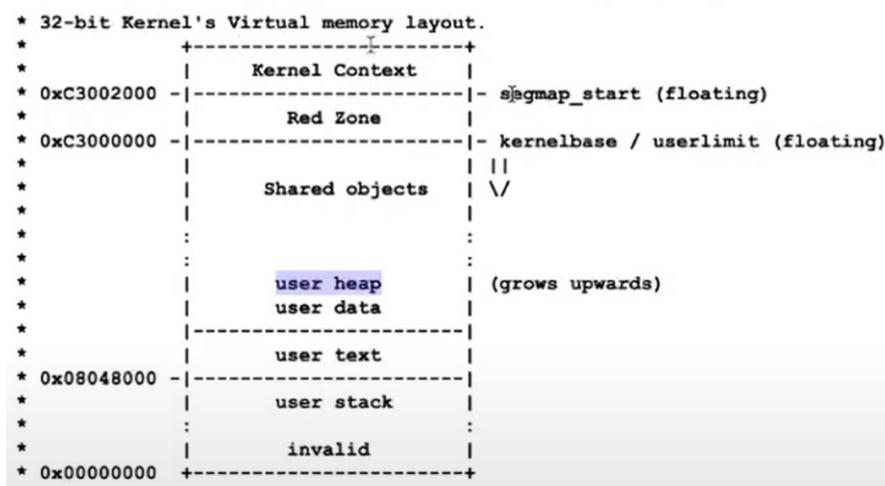
Куча - динамический сегмент. В куче может располагаться environment? Environment - не только переменные

## ▼ АДРЕСНОЕ ПРОСТРАНСТВО ПРОЦЕССА (РИС)

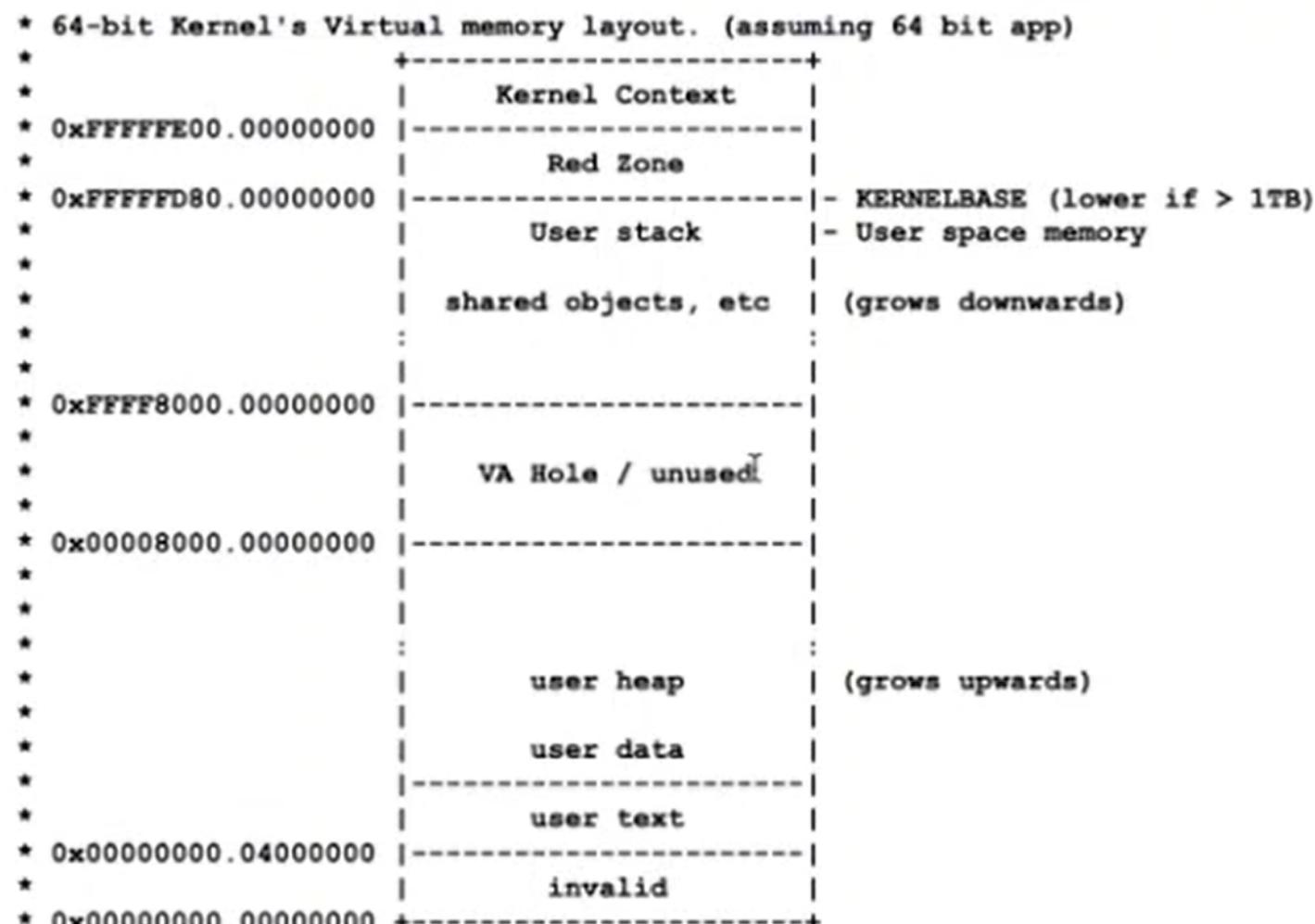
- В самом низу - защитная область, к ней нельзя обращаться (0-вой указатель не может указывать не на что валидное). Размер около 8мб
- Стек растет вниз
- Над стеком сегмент текст - машинный код программы
- Сегмент данных и куча. По умолчанию malloc выделяет память путем расширения сегмента данных, куча растет вверх
- Пользовательские сегменты (под верхней границей памяти) - динамические сегменты, создаваемые пользователем
- Red zone
- Kernel context 6мб - часть ядра, минимально необходимая для системных вызовов (не используется, пока не вызван системный вызов)
- Неиспользованные области

## Адресное пространство процесса x86

<http://cvs.opensolaris.org/source/xref/onv/onnv-gate/usr/src/uts/i68pc/os/startup.c>



- Стек находится под Red Zone, растет вниз
- Под ним динамические сегменты



## Книга Essential System Administration

### ▼ BOOT PROCESS

## BOOT PROCESS

1. После непродолжительного периода времени ядро ОС запускает **демон системы инициализации** (`/sbin/init`) является первым демоном, запущенным в рамках системы.  
Одним из первых действий `init` является проверка целостности локальных файловых систем, начиная с корневой файловой системы и других важных файловых систем, таких как `/usr`.
2. После чтения бинарника `/sbin/init` происходит чтение `/sbin/inittab` - конфигурационный файл `init`. В нем содержится информация об уровнях исполнения `rc` (у нас 3) (+ мб о том, как себя вести, когда `runlevel` меняется).
3. После того как `init` узнал `runlevel`, в `inittab` прописано что нужно запустить `rc(n)`. `rc(n)` запускает скрипт (сценарий), расположенный в `etc/rc3.d`

▼ С помощью сценариев инициализации системы выполняется следующее:

1. \*Сценарии инициализации системы обычно выполняют несколько предварительных действий, прежде чем приступить к загрузке системы. К ним относятся определение функций и локальных переменных, которые могут использоваться в сценарии, и настройка среды исполнения сценария, часто начиная с определения переменных среды `HOME` и `PATH`\*  
2. Проверка целостности файловых систем, традиционно с помощью утилиты `fsck`  
3. Монтирование локальных дисков  
4. Назначение и инициализация областей подкачки  
5. Выполнение действий по очистке файловой системы: проверка дисковых квот, сохранение файлов восстановления редактора и удаление временных файлов в `/tmp` и других местах.  
6. Запуск системных серверных процессов (демонов) для таких подсистем, как печать, электронная почта, бухгалтерский учет, регистрация ошибок и `cron`.  
7. Запуск сетевых демонов и монтирование удаленных дисков  
8. Включение входа пользователей в систему, обычно путем запуска процессов `getty` и/или графического интерфейса входа в систему на системной консоли (например, `xdm`) и удаления файла `/etc/nologin`, если он есть.
4. Init (в SVR4) запускает контроллер сервисов `/usr/lib/saf/sac` - он стартует, когда машина переводится в многопользовательский режим. Читает командный файл `/etc/saf/_sactab` и запускает **менеджера граф дисплеев - gdm** и **сервер удаленного доступа sshd** (устанавливает переменные `TERM` и `TZ`)  
И `./usr/lib/saf/ttymon` обслуживает терминалы. Он устанавливает переменную среды `TERM` (тип терминала) и определяет активность терминального порта и выдает приглашение входа в систему. После ответа на приглашение входа в систему, `ttymon` запускает программу `/bin/login` среду для дочернего процесса. Пример такого случая — программа `login`, которая инициализирует новую командную оболочку. Обычно программа `login` создает определенное окружение с небольшим количеством переменных и позволяет нам через файл начального запуска командной оболочки добавить свои переменные окружения при входе в систему.  
. `login` проверяет регистрационное имя пользователя и его пароль. При успешной проверке, `login` запускает поверх себя командный интерпретатор `shell`, определенный в файле `passwd` или в другой БД учетных записей. `login` производит установку следующих переменных среды исполнения:  
`HOME`=шестое поле в файле паролей  
`LOGNAME`=регистрационное имя пользователя  
`MAIL`=`/var/mail/[регистрационное имя пользователя]`  
`PATH`=`/usr/bin`  
`SHELL`=седьмое поле файла паролей (установка происходит, только если 7-ое поле не равно нулю)  
. После того как `login` запускает поверх себя `shell`, то `shell` сначала читает команды из файла `/etc/profile`, а затем из файла `$HOME/.profile`, если этот файл существует. Затем `shell` выдает приглашение для ввода команды. После этого `shell` интерпретирует команды пользователя и выполняет их как порожденные процессы.  
Среда инициализируется процессом `init` и модифицируется при входе пользователя в систему программами `login` или `gdm`. При удаленном входе в систему при помощи `ssh(1)` или `XDMCP` также могут передаваться переменные среды. Так, по умолчанию, `ssh` передает переменные среды `TERM` (тип терминала) и `TZ` (временная зона, часовой пояс), так что удаленная сессия живет в соответствии с настройками часового пояса той системы, за клавиатурой которой сидит пользователь.

Shell изменяет свою среду исполнения при интерпретации входного файла `/etc/profile`. Этот файл содержит все команды и установки переменных среды, которые системный администратор хочет выполнить для каждого входящего в систему. Если личная директория содержит файл `.profile`, shell читает команды из этого файла и модифицирует среду конкретного пользователя. Поскольку все процессы терминальной сессии являются потомками входного shell, то все они наследуют сделанные в `/etc/profile` и `$HOME/.profile` настройки.

5. `gdm` играет роль процессов `ttymon` и `login` (`ttymon` устанавливает переменную `TERM`). `gdm` запускает сервер X Window, устанавливает с ним соединение и выдает окно с запросом входа пользователя. При успешном входе `gdm` устанавливает переменную `DISPLAY=имя дисплея X Window` (нужна для соединения с сервером)
6. `gdm` запускает командный интерпретатор shell, который интерпретирует файл `/etc/gdm/Xsession`.

Этот файл считывает стандартные стартовые файлы shell `/etc/profile` и `$HOME/.profile`, так что все настройки переменных среды, которые сделали администратор системы и пользователь, также загружаются. Наконец, `Xsession` запускает менеджер графических сессий `gnome-session(1)`, который и запускает, собственно, графическую пользовательскую среду

#### ⇒ Гипотезы:

1. Хранятся в памяти процесса `gdm` (Графический вход в систему происходит аналогично тому, как вход через терминал, только роль `ttymon` и `login` играет программа X Display Manager)  
потом передаются в стартовые файлы профиля (Затем `gdm` запускает командный интерпретатор shell, который интерпретирует файл `/etc/gdm/Xsession`. Этот файл считывает стандартные стартовые файлы shell `/etc/profile` и `$HOME/.profile`, так что все настройки переменных среды, которые сделали администратор системы и пользователь, также загружаются)
  - Среда инициализируется процессом `init` и модифицируется при входе пользователя в систему программами `login` или `gdm/xdm`

## ▼ СТАДИИ ЗАГРУЗКИ UNIX

1. Базовое определение оборудования (память, диск, клавиатура, мышь и т.п.).
2. Выполнение программы инициализации системы прошивки (происходит автоматически).
3. Поиск и запуск программы начальной загрузки (с помощью программы загрузки микропрограммы), обычно из заранее определенного места на диске. Эта программа может выполнять дополнительные проверки оборудования перед загрузкой ядра.
4. Поиск и запуск ядра Unix (с помощью программы начальной загрузки). Файл образа ядра для выполнения может быть определен автоматически или посредством ввода в программу загрузки.
5. Ядро само инициализируется, а затем выполняет окончательные высокоДуровневые проверки оборудования, загружая драйверы устройств и/или модули ядра по мере необходимости.
6. Ядро запускает процесс инициализации, который, в свою очередь, запускает системные процессы (демоны) и инициализирует все активные подсистемы. Когда все готово, система начинает принимать логины пользователей.

## ▼ ЯДРО

Ядро — это часть операционной системы Unix, которая работает все время, когда система работает. Сам исполняемый образ ядра, условно именуемый unix (системы на основе System V). Обычно он хранится в корневом каталоге или связан с ним.

Как только управление передается ядру, оно подготавливается к запуску системы, инициализируя свои внутренние таблицы, создавая структуры данных в памяти с размерами, соответствующими текущим системным ресурсам и значениям параметров ядра. Ядро также может выполнять диагностику оборудования, которая является частью процесса загрузки, а также устанавливать загружаемые драйверы для различных аппаратных устройств, присутствующих в системе. Когда эти подготовительные действия завершены, ядро создает другой процесс, который запустит программу инициализации как процесс с PID 1.

## ▼ INIT

`Init` является предком всех последующих процессов Unix и *прямым родителем оболочек входа пользователей*.

Одним из первых действий `init` является проверка целостности локальных файловых систем, начиная с корневой файловой системы и других важных файловых систем, таких как `/usr`. Поскольку ядро и сама программа инициализации находятся в корневой файловой системе (или иногда в файловой системе `/usr` в случае `init`), как одна из них может работать до того, как соответствующая файловая система будет проверена:

- Иногда копия ядра находится в загрузочном разделе корневого диска, а также в корневой файловой системе.
- В качестве альтернативы, если исполняемый файл из корневой файловой системы успешно начинает выполняться, вероятно, можно с уверенностью предположить, что с файлом все в порядке. В случае `init` есть несколько возможностей. В System V корневая файловая система монтируется только для чтения до тех пор, пока она не будет проверена, и `init` перемонтирует ее для чтения и записи

## ▼ ДОП ДЕЙСТВИЯ INIT'А:

1. Проверка целостности файловых систем, традиционно с помощью утилиты `fsck`
2. Монтирование локальных дисков
3. Назначение и инициализация областей подкачки
4. Выполнение действий по очистке файловой системы: проверка дисковых квот, сохранение файлов восстановления редактора и удаление временных файлов в `/tmp` и других местах.
5. Запуск системных серверных процессов (демонов) для таких подсистем, как печать, электронная почта, бухгалтерский учет, регистрация ошибок и `cron`.
6. Запуск сетевых демонов и монтирование удаленных дисков
7. Включение входа пользователей в систему, обычно путем запуска процессов `getty` и/или графического интерфейса входа в систему на системной консоли (например, `xdm`) и удаления файла `/etc/nologin`, если он есть.

Эти действия задаются и выполняются с помощью сценариев *инициализации системы*, программ оболочки, традиционно хранящихся в `/etc` или `/sbin` или их подкаталогах и выполняемых `init` во время загрузки. После завершения этих действий пользователи могут войти в систему. На этом процесс загрузки завершен, и говорят, что система находится в многопользовательском режиме.

## ▼ ПРОЦЕСС ЗАГРУЗКИ

1. `init` управляет процессом загрузки в многопользовательском режиме. `init` запускает любые сценарии инициализации, для которых он был разработан, а структура программы `init` определяет набор сценариев инициализации для этой версии Unix: как называются сценарии, где они расположены в файловой системе, последовательность в которых они выполняются, ограничения, наложенные на программистов сценариев, предположения, в которых они работают, и так далее.

### Подготовка

Сценарии инициализации системы обычно выполняют несколько предварительных действий, прежде чем приступить к загрузке системы. К ним относятся определение функций и локальных переменных, которые могут использоваться в сценарии, и настройка среды исполнения сценария, часто начиная с определения переменных среды `HOME` и `PATH`:

```
HOME=/; export HOME  
PATH=/bin:/usr/bin:/sbin:/usr/sbin; export PATH
```

### Подготовка файловой системы

Подготовка файловой системы к использованию — это первый и наиболее важный аспект процесса многопользовательской загрузки. Он естественным образом делится на две фазы: монтирование корневой файловой системы и других жизненно важных системных файловых систем (таких как `/usr`) и обработка оставшейся части локальных файловых систем.

### Проверка и монтирование корневой файловой системы

#### Подготовка других локальных файловых систем

`fsck` command

После того, как все локальные файловые системы проверены (или определено, что они не нужны), их можно смонтировать с помощью команды `mount`, как в этом примере из системы BSD

## **Запуск серверов и инициализация локальных подсистем**

Когда все необходимые системные устройства готовы, можно запускать важные подсистемы, такие как электронная почта, печать и учет. Большинство из них полагаются на демонов (серверные процессы). Эти процессы запускаются автоматически одним из сценариев загрузки. В большинстве систем чисто локальные подсистемы, которые не зависят от сети, обычно запускаются до инициализации сети, а подсистемы, которым необходимы сетевые средства, запускаются позже.

...

## **Подключение к сети**

После запуска базовой сети можно запустить другие зависящие от нее службы и подсистемы. В частности, удаленные файловые системы можно смонтировать с помощью такой команды, которая монтирует все удаленные файловые системы, перечисленные в файле конфигурации файловой системы системы

## **Уборка**

## **Допуск пользователей в систему**

### **Initialization Files on System V Systems**

The system initialization scripts on a System V-style system are much more numerous and complexly interrelated than those under BSD. They all revolve around the notion of the current system run level, a concept to which we now turn.

#### **Initialization files overview**

System V-style systems organize the initialization process in a much more complex way, using three levels of initialization files:

- */etc/inittab*, which is init's configuration file.
- A series of primary scripts named *rcn* (where *n* is the run level), typically stored in */etc* or */sbin*.
- A collection of auxiliary, subsystem-specific scripts for each run level, typically located in subdirectories named *rcn.d* under */etc* or */sbin*.
- In addition, some systems also provide configuration files that define variables specifying or modifying the functioning of some of these scripts.

#### **The init configuration file**

As we've seen, top-level control of changing system states is handled by the file */etc/inittab*, read by *init*. This file contains entries that tell the system what to do when it enters the various defined system states.

## **▼ КОНФИГУРАЦИОННЫЙ ФАЙЛ INIT'А**

▼ Конфигурационный файл init'a /inittab логически состоит из 7 секций:

1. Первый раздел, состоящий из одной записи, устанавливает уровень запуска по умолчанию, которым в данном случае является сетевой многопользовательский режим (уровень 3).
2. Второй раздел содержит процессы, запускаемые при загрузке системы. В примере файла это состоит из запуска сценариев предварительной загрузки /etc/bcheckrc и /etc/brc (в Солярисе заменяются скриптом /sbin/rcS), обычно используемых в системах System V, в дополнение к структуре rc(n). Основная функция сценария bcheckrc — подготовить корневую файловую систему и другие важные файловые системы, такие как /usr и /var. Оба сценария могут завершиться до того, как init перейдет к следующей записи inittab. (Есть программа /etc/ksh.kshrc )
3. В третьем разделе примера файла inittab указаны команды, которые должны выполняться всякий раз, когда система отключается, либо во время выключения и остановки системы (для уровня запуска 0), либо во время перезагрузки (уровень запуска 6). В обоих случаях выполняется сценарий /etc/rc0, и init ожидает его завершения, прежде чем продолжить.
  - ▶ Четвертый раздел, озаглавленный «изменения уровня выполнения», определяет команды, которые должны выполняться, когда система переходит в состояния 1, 2 и 3.
1. В пятом разделе указаны команды для запуска (после rc0), когда система переходит на уровни запуска 0 и 6. В обоих случаях init запускает команду uadmin, которая инициирует завершение работы системы.
2. Шестой раздел инициализирует терминальные линии системы с помощью процессов getty .
3. Иллюстрирует использование специального уровня выполнения a. Эта запись используется только тогда, когда системный администратор выполняет команду telinit a, после чего запускается сценарий start\_acct. Уровни выполнения a, b и c доступны для определения по мере необходимости.

#### The rc(n) initialization scripts

Таким образом, когда система переходит в состояние 0, init запускает rc0 (как указано в файле inittab), который, в свою очередь, запускает сценарии в rc0.d.

#### Solaris initialization scripts

Solaris использует стандартную схему сценария загрузки System V. Скрипт rcS (в /sbin) заменяет bcheckrc, но выполняет те же функции. Solaris использует отдельные сценарии rcn для каждого уровня выполнения от 0 до 6. Существуют отдельные каталоги rcn.d для уровней выполнения от 0 до 3 и S.

## Оболочка

The file `/etc/passwd` is the system's master list of information about users, and every user account has an entry within it. Each entry in the password file is a single line having the following form: `username:x:UID:GID:user information:home-directory:login-shell`

### `login shell`

The program used as the command interpreter for this user. Whenever the user logs in, this program is automatically started. This is usually one of `/bin/sh` (Bourne shell), `/bin/csh` (C shell), or `/bin/ksh` (Korn shell).<sup>‡</sup> There are also alternative shells in wide use, including `bash`, the Bourne-Again shell (a Bourne shell-compatible replacement with many C shell- and Korn shell-like enhancements), and `tcsh`, an enhanced C shell-compatible shell.

Here is a typical entry in `/etc/passwd`:

```
chavez:x:190:100:Rachel Chavez:/home/chavez:/bin/tcsh
```

This entry defines a user whose username is `chavez`. Her UID is 190, her primary group is group 100, her full name is Rachel Chavez, her home directory is `/home/chavez`, and she runs the enhanced C shell as her command interpreter.

## User Environment Initialization Files - |After adding a user to the `/etc/passwd` file, you must create a home directory for the user

Next, you should give the user copies of the appropriate initialization files for the shell and graphical environment the account will run (as well as any additional files needed by commonly used facilities on your system).

The various shell initialization files are:

Bourne shell	<code>.profile</code>
C shell	<code>.login</code> , <code>.logout</code> , and <code>.cshrc</code>
Bourne-Again shell	<code>.profile</code> , <code>.bash_profile</code> , <code>.bash_login</code> , <code>.bash_logout</code> , and <code>.bashrc</code>

These files must be located in the user's home directory. They are all shell scripts (each for its respective shell) that are executed in the standard input stream of the login shell, as if they had been invoked with `source` (C shells) or `.` (sh, bash, or ksh). The `.profile`, `.bash_profile`, `.bash_login`, and `.login` initialization files are executed at

`login.*`, `.cshrc`, `.tcshrc`, `.bashrc`, and `.kshrc` are executed every time a new shell is spawned. `.logout` and `.bash_logout` are executed when the user logs out.

As administrator, you should create standard initialization files for your system and store them in a standard location. Conventionally, the directory used for this purpose is `/etc/skel`, and most Unix versions provide a variety of starter initialization files in this location. These standard initialization files and the entire directory tree in which they are kept should be writable only by `root`.

## Sample login initialization files

The `.*login` or `.*profile` files are used to perform tasks that only need to be executed upon login, such as:

- Setting the search path
- Setting the default file protection (with `umask`)
- Setting the terminal type and initializing the terminal
- Setting other environment variables
- Performing other customization functions necessary at your site

## Systemwide initialization files

Для пользователей оболочки Bourne, Bourne-Again и Korn файл `/etc/profile` служит общесистемным файлом инициализации, который выполняется перед файлом инициализации личного входа пользователя. В нем почти всегда определена переменная PATH; поэтому он применяется к пользователям без явных переменных PATH, установленных в их `.profile`. Иногда здесь также указывается `umask` по умолчанию. Вот простой файл `/etc/profile`, разработанный для оболочки bash, адаптированный из системы Red Hat Linux; мы аннотировали его комментариями:



