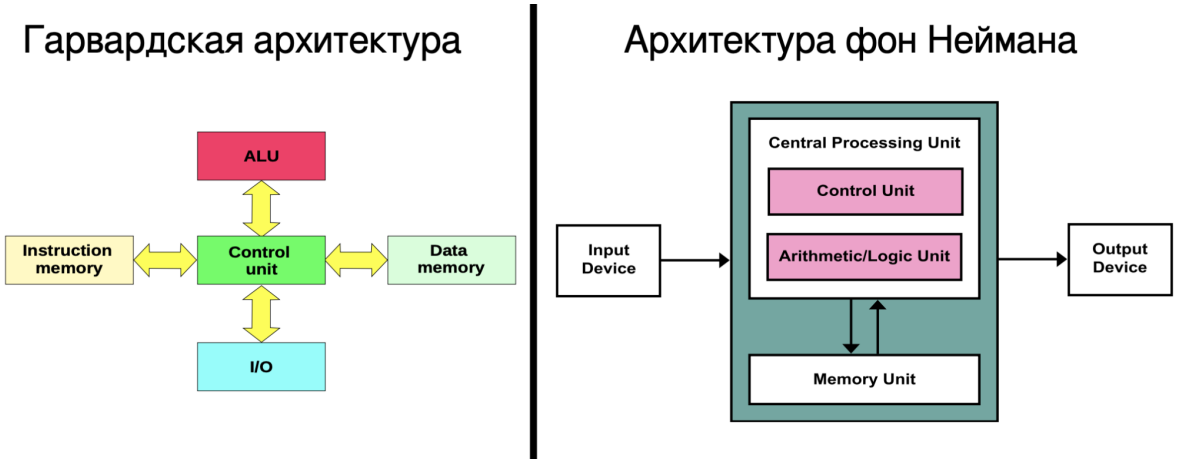
1. Архитектура компьютерных систем. Архитектура Фон-Ндская архитектура. Принципы архитектуры Фон-Неймана и Гарваймана. Архитектуры NUMA и UMA.



Существует два основных вида архитектуры: фон Неймана и Гарвардская.
В то фон Неймана данные и команды хранятся в одной и той же памяти
В Гарвардской - отдельная память для команд, отдельная для хранения данных

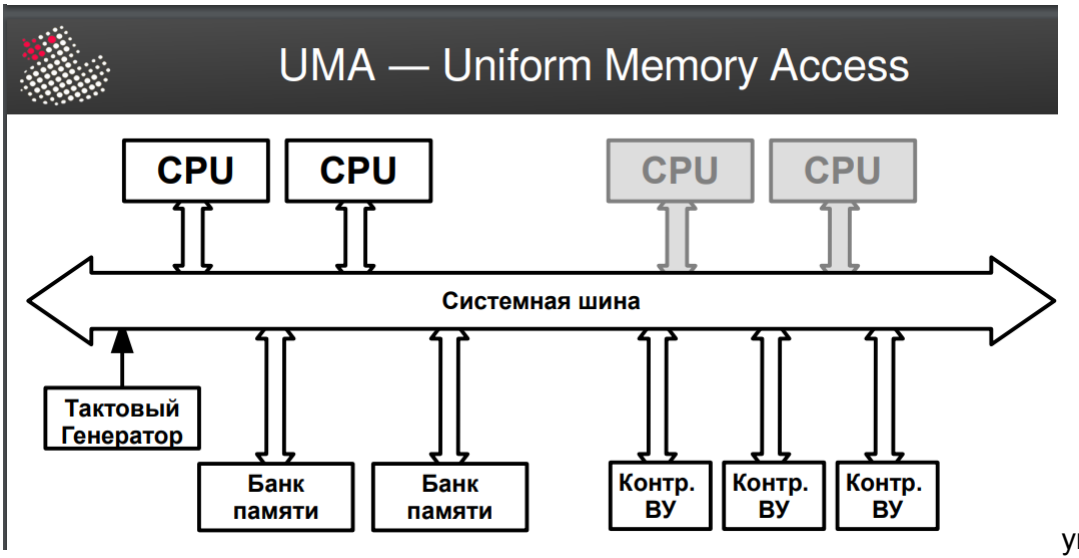
- Принципы Фон-Неймана:
  - однородность памяти (команды и данные хранятся вместе, внешне неразличимы)
  - принцип адресности(у каждой ячейки памяти есть свой адрес, к которому имеет доступ процессор)
  - принцип программного управления(вычисления представлены в виде программы, состоящей из последовательности команд)
  - принцип двоичного кодирования(данные и команды кодируем нулями и единичками)

- Принципы Гарвардской
  - отдельно данные, отдельно инструкции
  - канал инструкций и канал данных также отличаются

С точки зрения пользовательских штук две штуки UMA и NUMA:

- UMA (Uniform Memory Access / однородный доступ к памяти): Все устройства одного ранга и все устройства имеют одинаковый доступ к памяти, а значит

время доступа для всех одинаково. Задержка одинакова и тд



уп.

Плюсы:

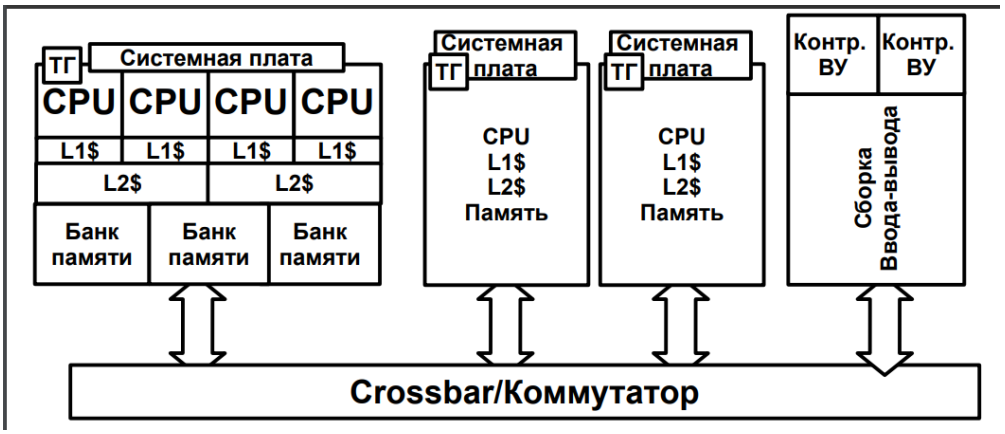
- одинаковое время доступа к памяти
- простота реализации

Минусы:

- Плохая масштабируемость (с увеличением количества процессов возникают конфликты и конкуренция за доступ к памяти, а это замедляет работу системы)

Работает нормально только до двух процессоров.

- NUMA (Non Uniform Memory Access / неоднородный доступ к памяти)



- все системные ресурсы расположены на отдельных системных платах
- имеется локальная память(расположенная “ближе” к процессору)
- общая память и устройства ввода-вывода доступны через коммутатор, координирующий взаимодействие компонентов
- Используется в серверах
- адресное пространство общее для всех процессов!!!

Плюсы:

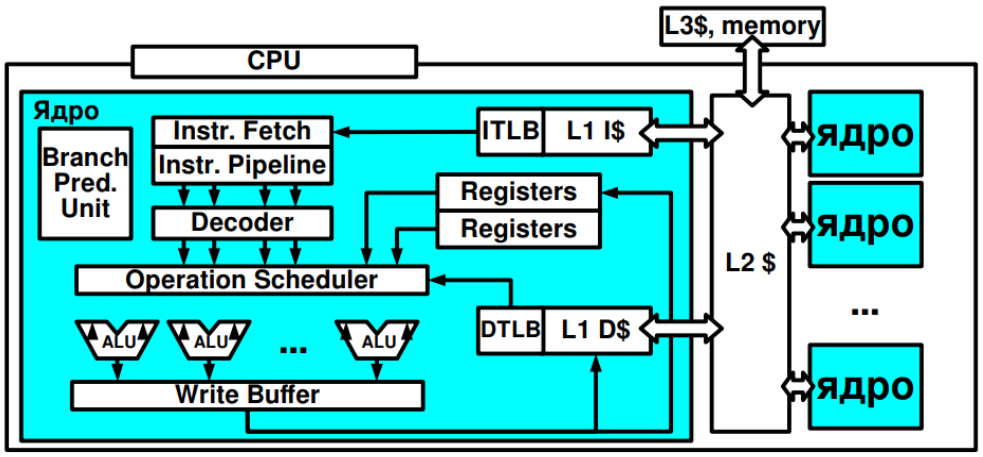
- возможность замены практически любого компонента во время работы системы (hot swap / горячая замена)
- Хорошая масштабируемость
- Надежность и отказоустойчивость

Минусы:

- требовательна к операционной системе(она должна поддерживать все эти фишки)
- важно учитывать время доступа и расположение данных в памяти(так как время доступа зависит от того, где данные расположены)

2. Общая организация процессора, памяти, организация вычислений.

Организация процессора



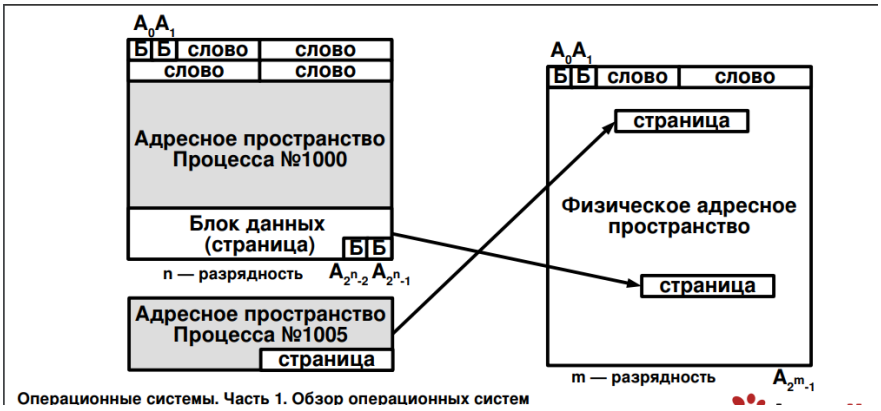
- Кэш второго уровня (L2 $): кэш общий для всех ядер
- Кэш первого уровня (L1$): разделен на кэш с данными (L1 D$) и кэш с инструкциями (L1 I$), оба имеют буфер TLB, это помогает параллельно читать данные и инструкции
- Буфер ассоциативной трансляции (TLB): хранит записи, которые помогают быстрее найти записи внутри кэша или понять есть они в кэше этого уровня или нет (ускоряет трансляцию адреса виртуальной памяти в адрес физической памяти)
- Hyper-threading (два блока Registers), суть технологии hyper-threading: у процессора есть два набора регистров, что позволяет быстрее переключать контекст двух потоков. Для операционной системы это выглядит как два логических процессора.
- предсказатель переходов (branch prediction unit): в случае когда выполняется условие в условном переходе, загруженные команды оказываются ненужными, а значит конвейер встанет, чтобы избежать этого создан предсказатель переходов, который дает понять какие команды грузить.
- Вычислительный конвейер (Упомянуто в лекции криво, лучше почитать вики или тут)

- параллельность циклов исполнения и выборки команды за счет множества АЛУ, позволяющих распараллелить очередь операций
- конвейер - получает команду с устройства выборки команды, декодирует и отдает планировщику операций
- планировщик - решает пустить пришедшие команды параллельно или нет и пустить ли вообще команду на выполнение, в зависимости от наличия данных
- branch pred unit - предсказатель перехода, для оптимизации загрузки команд (чтобы конвейер не останавливался)

- write buffer нужен т. к. скорость записи в память может не поспевать за ходом исполнения команд
- есть обычные алу а есть специфичные (для плавающей точки, для векторных операций и т.п)

Есть еще картинка из Столлингса попроще: СК,РК,АЛУ, РА,РБ,регистр адреса ввода-вывода, регистр буфера ввода-вывода

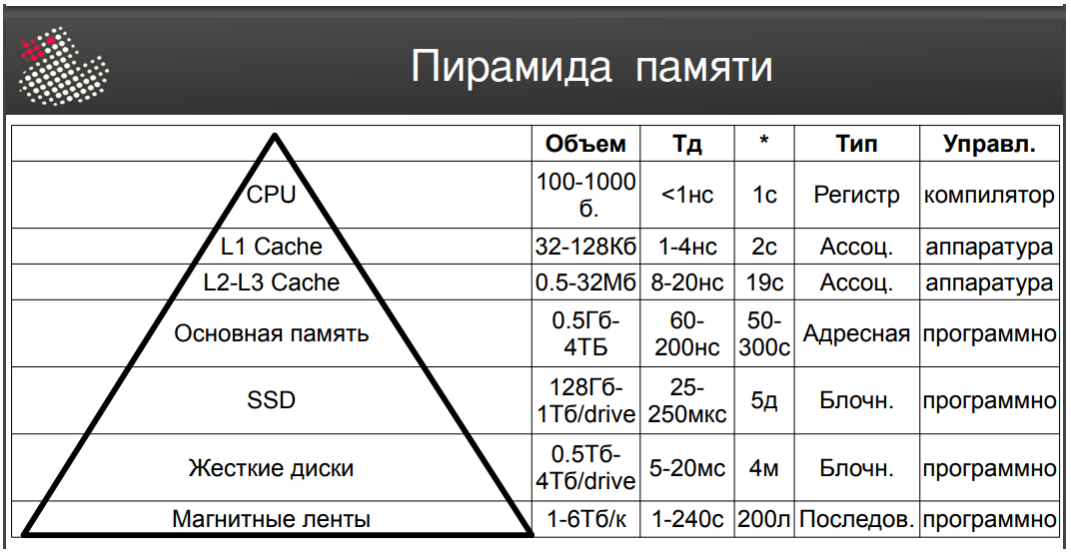
Организация памяти



- Виртуальная память – способ организации памяти, при котором каждый процесс имеет собственное адресное пространство,думая, что у него есть своя независимая память.
- Адреса виртуальной памяти преобразуются в физические при помощи таблиц трансляций.

- Единица доступа к информации – байт. Данные объединены в страницы (размер обычно 4 Кб).
- Large Pages – страницы с большим объемом данных (к примеру 1 Мб)

Преимущество: уменьшение области трансляции при выделении адресного пространства, релокация, swapping, защита



	Объем	Тд	*	Тип	Управл.
CPU	100-1000 б.	<1нс	1с	Регистр	компилятор
L1 Cache	32-128Кб	1-4нс	2с	Ассоц.	аппаратура
L2-L3 Cache	0.5-32Мб	8-20нс	19с	Ассоц.	аппаратура
Основная память	0.5Гб-4ТБ	60-200нс	50-300с	Адресная	программно
SSD	128Гб-1Тб/drive	25-250мкс	5д	Блочн.	программно
Жесткие диски	0.5Тб-4Тб/drive	5-20мс	4м	Блочн.	программно
Магнитные ленты	1-6Тб/к	1-240с	200л	Последов.	программно

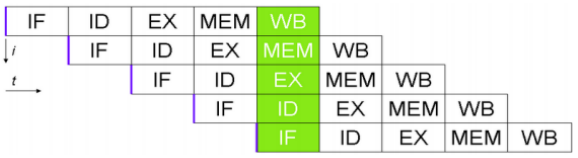
Память внутри компа организована иерархически. Самая быстрая и дорогая память наверху, снизу наибольший объем. Сделано так, чтобы к более медленной памяти было меньше обращений.

Тд(время доступа) – интервал между первым запросом доступа к памяти и первыми данными оттуда

Виды управления:
- Компилятор: управляется при помощи ключей компиляции (уровни оптимизации)
- Аппаратура: поиск осуществляется при помощи схем в ассоциативной памяти (TLB)
- Программно

Организация вычислений

- Ядро выполняет каждую команду последовательно. Цикл команды:
  - Выборка команды (Instruction Fetch)
  - Декодирование инструкций (Instruction Decode)
  - Исполнение (Exectution)
  - Чтение памяти (MEM)
  - Запись (Write Back)



Конвейерная обработка команд. Следующая команда начинает выполняться, пока не закончила выполнение предыдущая.

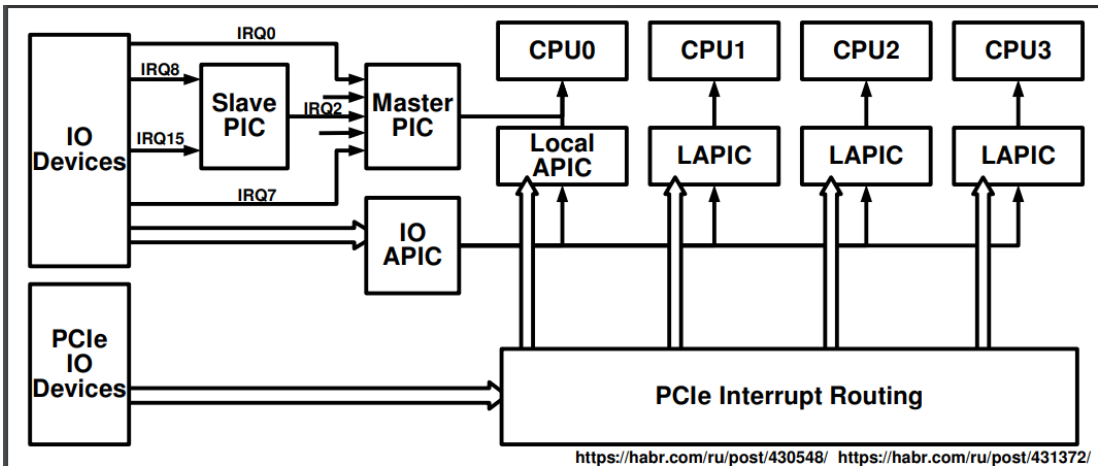
3. Организация прерываний, типы прерываний, контроллер прерываний.

Прерывание (англ. interrupt) — сигнал от программного или аппаратного обеспечения, сообщающий процессору о наступлении какого-либо события, требующего немедленного внимания. Прерывание извещает процессор о наступлении высокоприоритетного события, требующего прерывания текущего кода, выполняемого процессором. Процессор отвечает приостановкой своей текущей активности, сохраняя свое состояние и выполняя функцию, называемую обработчиком прерывания (или программой обработки прерывания), которая реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

- Могут быть вызваны самой программой или сигналом с устройств IO.
- Более высокий приоритет прерываний могут прерывать более низкие.
- Выполняются в конце цикла команды
- Могут быть вложенными. Есть ограничения по вложенности обработчика в зависимости от архитектуры.

КОНТРОЛЁР ПРЕРЫВАНИЙ

Контроллер прерываний (англ. Programmable Interrupt **Controller**, PIC) — микросхема или встроенный блок процессора, отвечающий за возможность последовательной обработки запросов на прерывание от разных устройств



Надо читать всю статью с [хабра]

Слайд интерпретируется так:

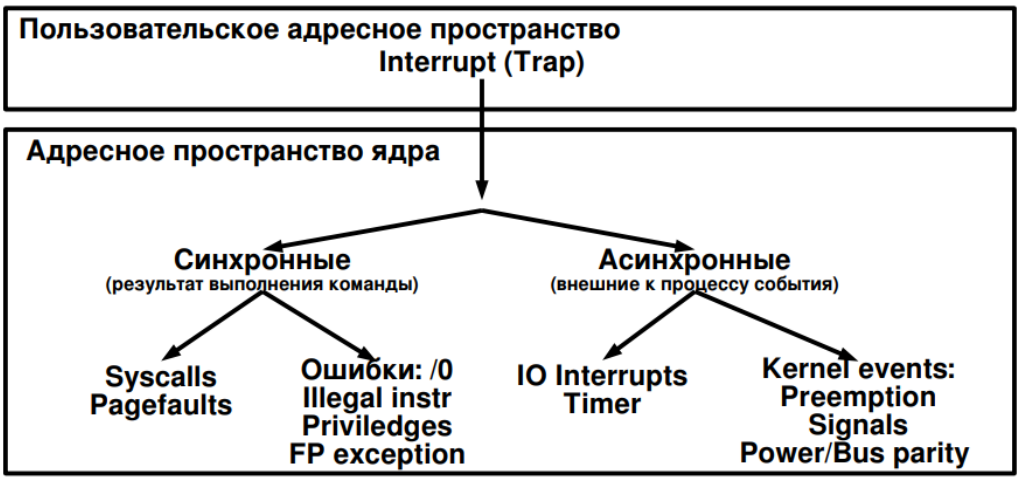
1. slave-master PIC - был придуман для увеличения линий прерываний
2. IO APIC, LAPIC - был придуман для поддержки многопроцессорности
3. PCIe_device -> {PCIe_Interrupt_Routing} -> Local_APIC -> CPU - пришло с появлением PCIe, информация о прерывании пишется (почти) напрямую в память LAPIC. {PCIe_Interrupt_Routing} - обобщение представляющее роутинг.

---.

Разные подходы:
-запрет повторных прерываний, пока обрабатываются предыдущие (последовательная обработка прерываний).
-обработка прерываний по уровню приоритета

(схема для x86 проца)
В контроллере прерываний есть несколько процессоров. У каждого процессора свой локальный маршрутизатор/контроллер прерываний, обработчик прерываний. Есть устройства IO, посылающие запрос на прерывание, логика, занимающаяся обработкой, отвечающая за приоритеты и прочее, локальный контроллер посылает процессору запрос на прерывание.

ТИПЫ ПРЕРЫВАНИЙ:



асинхронные, или внешние (аппаратные) — события, которые исходят от внешних аппаратных устройств (например, периферийных устройств) и могут произойти в любой произвольный момент: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши. Факт возникновения в системе такого прерывания трактуется как запрос на прерывание (англ. Interrupt request, IRQ) — устройства сообщают, что они требуют внимания со стороны ОС;

синхронные, или внутренние — события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение стека, обращение к недопустимым адресам памяти или недопустимый код операции;

4. Типичные функции операционной системы. Интерфейсы ОС. Работа ОС как замена оператора ЭВМ.

Наиболее важной из системных программ является операционная система, которая скрывает от программиста детали аппаратного обеспечения и предоставляет ему удобный интерфейс для использования системы. Операционная система выступает в роли посредника, облегчая программисту и программным приложениям доступ к различным службам и возможностям.

Функции:

- Разработка программ
- Выполнение программ
- Доступ к устройствам ввода-вывода
- Контролируемый доступ к файлам
- Доступ к системе и системным ресурсам
- Обнаружение и обработка ошибок
- Учет использования и диспетчеризация ресурсов
- Предоставление ключевых интерфесов ОС:
  - ISA (Instruction Set Architecture) — Набор команд
  - ABI (Application Binary Interface) — Бинарный интерфейс приложения
  - API (Application Programming Interface) — Интерфейс прикладных программ

Если кратко, то это менеджмент доступа программ к ресурсам, а также упрощение разработки программ (использование API, как библиотеки низкого уровня)

Существуют редакторы, отладчики(программы-утилиты), которые не входят в состав ядра ОС, но поддерживаются ею, и помогают в разработке программ.

Контролируемый доступ к файлам состоит в том, что ОС определяет природу устройств и структуры их данных.

Структура системы команд (instruction set architecture — ISA). Определяет набор команд машинного языка, которые может выполнять компьютер. Этот интерфейс является границей между аппаратным и программным обеспечением. Обратите вни-

Бинарный интерфейс приложения (application binary interface — ABI). ABI определяет стандарт бинарной переносимости между программами. ABI определяет интерфейс системных вызовов операционной системы и аппаратных ресурсов и служб, доступных в системе через пользовательскую ISA.

Интерфейс прикладного программирования (application programming interface — API). API обеспечивает программе доступ к аппаратным ресурсам и службам, доступным в системе через пользовательскую ISA с библиотечными вызовами на языке высокого уровня. Обычно любые системные вызовы выполняются через библиотеки. Применение API обеспечивает легкую переносимость прикладного программного обеспечения на другие системы, поддерживающие тот же API, путем перекомпиляции.

До оператора ученые = программисты сами набирали программу, запускали и уходили на какое-то время, пока она не выполнит свою работу. Было расписание, по которому пользователи приходили со своими программами.
Потом компьютеры стали быстрее, стало сложно соблюдать такое расписание, появились операторы, которым отдавались программы и которые уже сами, по мере освобождения компьютера, запускали новую программу.

- В первых ЭВМ был только пульт управления
- Оператор должен был:
  - получить программу с данными от программиста;
  - подготовить программу к загрузке (н-р, с перфокарт);
  - загрузить программу и компилятор;
  - запустить программу на вычисление;
  - распечатку с результатами передать программисту.
- Минусы:
  - Наличие расписания машинного времени
  - Долгое время подготовки к работе

5. Пакетная обработка. Системный монитор.

Потом появился системный монитор
По сути весь доп. софт, который был нужен для работы программы, заранее загрузили в ЭВМ, плюс добавили прерывания, которые нужны, чтобы сигнализировать о том, что программа выполнилась, и планировщик, чтобы по этому сигналу запускать следующую программу. Программы все еще выполняются последовательно.

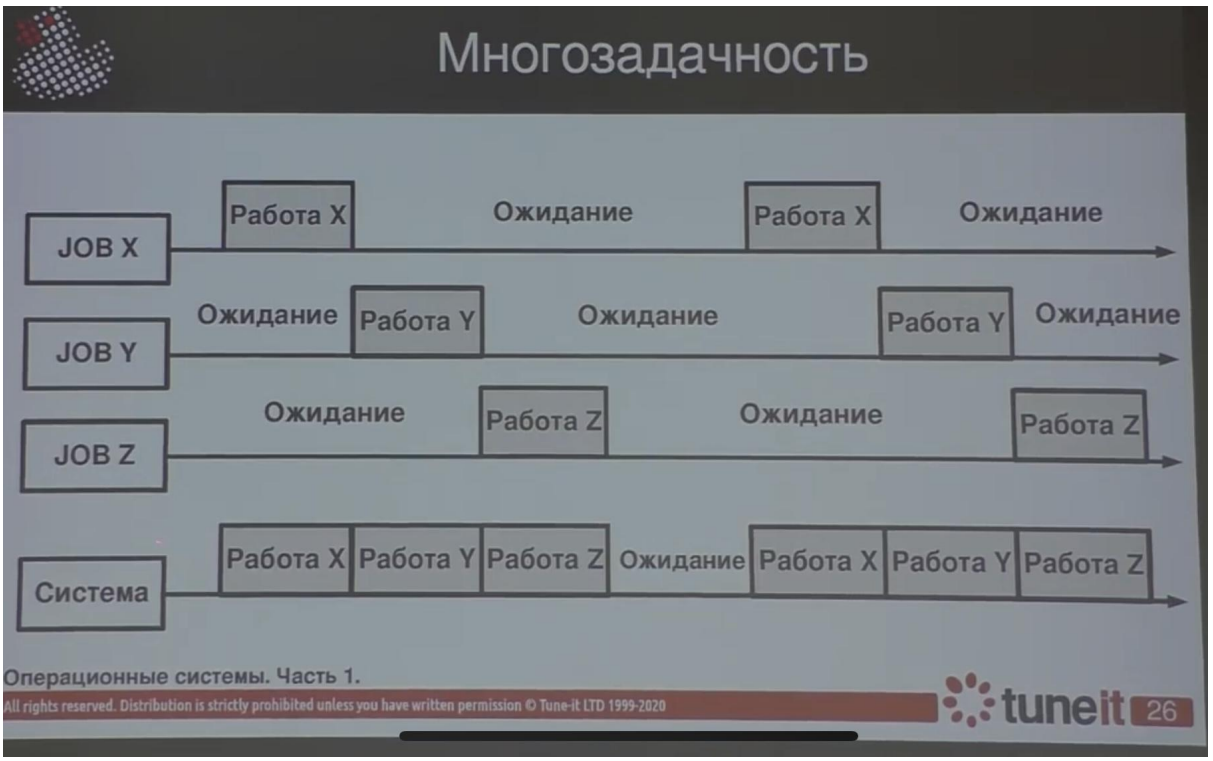
Центральная идея, лежащая в основе простых пакетных схем обработки, состоит в использовании программы, известной под названием **монитор** (monitor). Используя операционную систему такого типа, пользователь не имеет непосредственного доступа к машине. Вместо этого он передает свое задание на перфокартах или магнитной ленте оператору компьютера, который собирает разные задания в пакеты и помещает их в устройство ввода данных. Так они передаются монитору. Каждая программа составлена таким образом, что при завершении ее работы управление переходит к монитору, который автоматически загружает следующую программу.

- Машинное время дорогое, его простои необходимо минимизировать
- 1950 г., General Motors, IBM 701
- Наборы программ и данных передавались оператору и запускались

Драйверы устройств – осуществляют работу с устройствами (например, ввода-вывода)

6. Анализ общесистемной эффективности, как предусловие многозадачности. Многозадачность, как способ повышения системной эффективности. Системы разделения времени.

- Одно задание плохо загружает CPU:
  - Read (15 мкс) → Compute 100 instr (1 мкс) → Write (15 мкс)
  - Общая загрука CPU ~ 3.2%
- Давайте запустим много задач, и пока одни занимаются вводом-выводом, другие будут производить вычисления



- Хорошо бы исключить оператора и добавить пользователей!
  - Посадим юзеров за терминалы, пусть сами работают
  - Будем выдавать им часть времени процессора с использованием квантования времени (time slices)
- CTSS (CompatibleTime-Sharing System), MIT 1961, IBM 709
  - Выгрузка и загрузка задач
  - 32 пользователя
- Появились проблемы разделения ресурсов и защиты одних программ от других

Пользователей много, всем нужен компьютер, и заметили, что кучу времени процессор простаивает, потому что программе нужно получить данные, например, с диска. Решили сделать так, чтобы в это время на процессоре выполнялась другая программа, которая ничего не ждет. Концептуально ОС появилась уже тогда, в виде систем разделения времени, но это был по сути только скелет.

7. Процессы, проблемы современных процессов. Планирование выполнения процессов и управление ресурсами.

Процесс – экземпляр программы во время ее исполнения.

Процесс – это единица активности ОС, в которой существуют последовательные действия, текущее состояние и набор связанных ресурсов. Данное определение с точки зрения ОС.

статья про процессы

Структура процесса

- Исполняемая программа (процесс представляет собой программу)
- Набор потоков исполнения (внутри программы включено множество потоков исполнения, это по сути единица потребления процессора)
- Связанные структуры ядра (когда процесс создается внутри ОС, ядро должно построить структуры,области памяти, которые содержат описания ресурсов которые процесс потребляет)
- Адресное пространство -- совокупность всех допустимых адресов каких-либо объектов вычислительной системы — ячеек памяти, секторов диска, узлов сети и т. п., которые могут быть использованы для доступа к этим в определенном процессе
- Контекст исполнении (например, регистры)(состояние процесса - то, что нужно восстановить когда мы возвращаемся допустим из другого процесса к этому)
- Контекст безопасности (всякие suid/sgid в linux)
- Ресурсы (файлы и прочее) + структуры, связанные с ними структуры
- Динамические библиотеки

Проблемы современных процессов

- Защита памяти процесса (недетерминированное поведение программы, то есть непредсказуемое)
- Взаимные блокировки (deadlocks, starvation, livelocks)
- Проблемы синхронизации
- Взаимное исключение доступа к ресурсам

Динамическая взаимоблокировка (livelock)	Состояние, при котором два или более процессов постоянно изменяют свое состояние в ответ на изменения в другом процессе (или процессах) без какой-либо полезной работы. Это похоже на взаимоблокировку тем, что при этом отсутствует какой-либо прогресс, но отличается тем, что ни один процесс не заблокирован и не ждет чего-либо

Взаимное блокирование (deadlock[1]) можно определить как *перманентное* блокирование множества процессов, которые либо конкурируют в борьбе за системные ресурсы, либо сообщаются один с другим. Множество процессов оказывается взаимно блокиро-

Голодание (starvation)	Состояние, при котором процесс откладывается на неопределенное время, потому что предпочтение постоянно отдается другим процессам

Планирование выполнение процессов и управление ресурсами

- Равноправие (Пользователи должны получать ресурсы равноправно)
- Дифференциация отклика (необходимость снижения времени отклика)
- Общесистемная эффективность
- Планировщики процессов, дисков и прочее (Разные классы диспетчеризации (Time Sharing, interactives, real time, system, fair share (типа каждому пропорционально приоритету~количеству акций), fixed..)) I/O scheduling

8. Управление памятью, виртуальная память. Защита информации и безопасность ОС.

Управление памятью

1. Изоляция процессов (отслеживание, что ни один из процессов не смог изменить чужую память)
2. Управление выделением и освобождением памяти (программы должны динамически подключаться к памяти - т.е. надо в том числе уметь передать ей управление)
3. Поддержка модулей (возможность определять, создавать, уничтожать и менять размер)
4. Защита и контроль доступа (ОС должна следить каким образом различные пользователи могут осуществлять доступ к различным областям памяти)
5. Долгосрочное хранение - приложениям требуются средства, с помощью которых можно хранить информацию после выключения компьютера.
6. Страничный обмен (**Подкачка страниц** (англ. *paging*; иногда используется термин *swapping* от *swap*, /swɔp/) — один из механизмов виртуальной памяти, при котором отдельные фрагменты памяти перемещаются из оперативной памяти во вторичное хранилище (жёсткий диск или другой внешний накопитель), освобождая оперативку для загрузки других активных фрагментов памяти. Такими фрагментами в современных ЭВМ являются страницы памяти.). В винде: page file sys.

Виртуальная память обслуживается MMU и TLB, делает отдельное адресное пространство для каждого процесса и ядра.
Виртуа́льная па́мять (англ. *virtual memory*) — метод управления памятью компьютера, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, путём автоматического перемещения частей программы между основной памятью и вторичным хранилищем

+ Существуют невыгружаемые страницы.

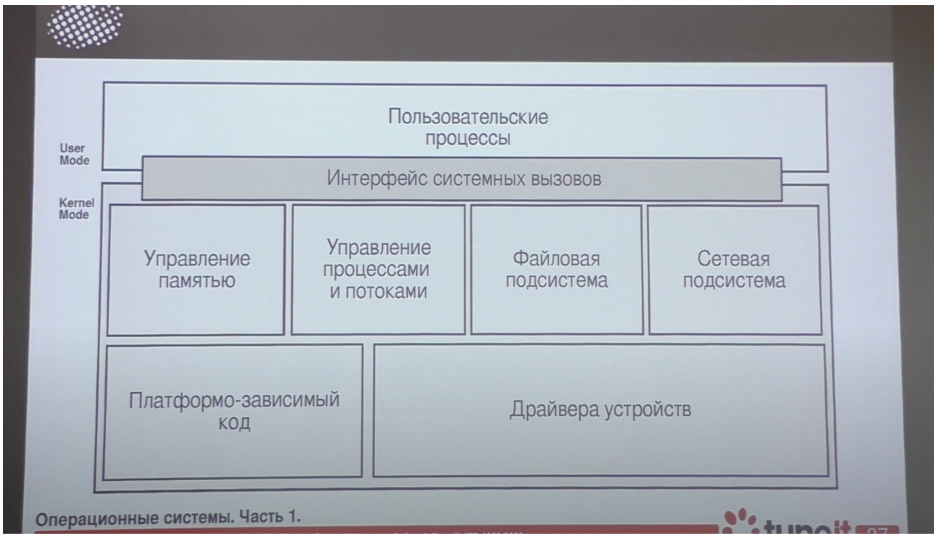
Защита информации и безопасность ОС. Все, что дается по безопасности: регламенты. Не факт, что будет работать, но есть процедуры (протоколы), соблюдение которых может помочь.

- Доступ к системе (защита от несанкционированного доступа)
- Конфиденциальность (невозможность получить доступ без авторизации)
- Целостность данных (защита данных от неавторизованного и нецелостного изменения)

- Аутентификация и авторизация (**Аутентификация** — процедура проверки подлинности, например проверка подлинности пользователя путем сравнения введенного им пароля с паролем, сохраненным в базе данных. **Авторизация**— предоставление определенному лицу или группе лиц прав на выполнение определенных действий.)

9. Структура ядра операционной системы. Архитектуры монолитного ядра, ядра динамически загружаемыми модулями и микроядра.

ЯДРО ОС



Есть пользовательский режим, есть режим ядра, есть интерфейс вызовов, где данные передаются между пользовательскими процессами и самой системой.
1)Монолитное ядро
Да

Монолитное ядро (monolithic kernel)	Большое ядро, содержащее практически всю операционную систему, включая планирование, файловую систему, драйверы устройств и управление памятью. Все функциональные компоненты ядра имеют доступ ко всем его внутренним структурам данных и процедурам. Как правило, монолитное ядро реализовано как единый процесс со всеми элементами, разделяющими одно и то же адресное пространство

Монолитное ядро (аналог мониторов); есть служебная программа, информация загружается в память и там работает; ядро нельзя изменить.

Сейчас в чистом виде не используется нигде, кроме как встроенные системы.

2)Модульное ядро — современная, усовершенствованная модификация архитектуры монолитных ядер операционных систем.
В отличие от «классических» монолитных ядер, модульные ядра, как правило, не требуют полной перекомпиляции ядра при изменении состава аппаратного обеспечения компьютера. Вместо этого модульные ядра предоставляют тот или иной механизм подгрузки модулей ядра, поддерживающих то или иное аппаратное обеспечение (например,драйверов). При этом подгрузка модулей может быть как динамической (выполняемой «на лету», без перезагрузки ОС, в работающей системе), так и статической (выполняемой при перезагрузке ОС после переконфигурирования системы на загрузку тех или иных модулей).

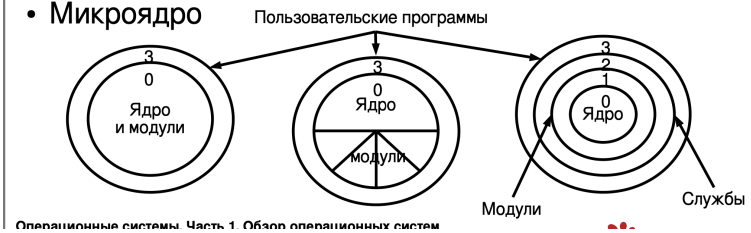
Команды для загрузки и удаления в линукс: rsmod, modprob, rmmod. Удобно включить/выключить не перегружая всю машину.

3)Микроядро

Микроядро (microkernel)	Небольшое привилегированное ядро операционной системы, обеспечивающее планирование процессов, управление памятью и службы связи и опирающееся на другие процессы для выполнения некоторых функций, традиционно связываемых с ядром операционной системы

Ядро выполняет базовые функции, а остальные функции осуществляются в виде сервисов и находятся на своих уровнях безопасности. Работает медленно за счет частого переключения контекстов. (пример: Windows NT)

- Монолитное ядро
- Ядро с динамически загружаемыми модулями
- Микроядро



10. Потоки исполнения, многопоточность, модели многопоточности.

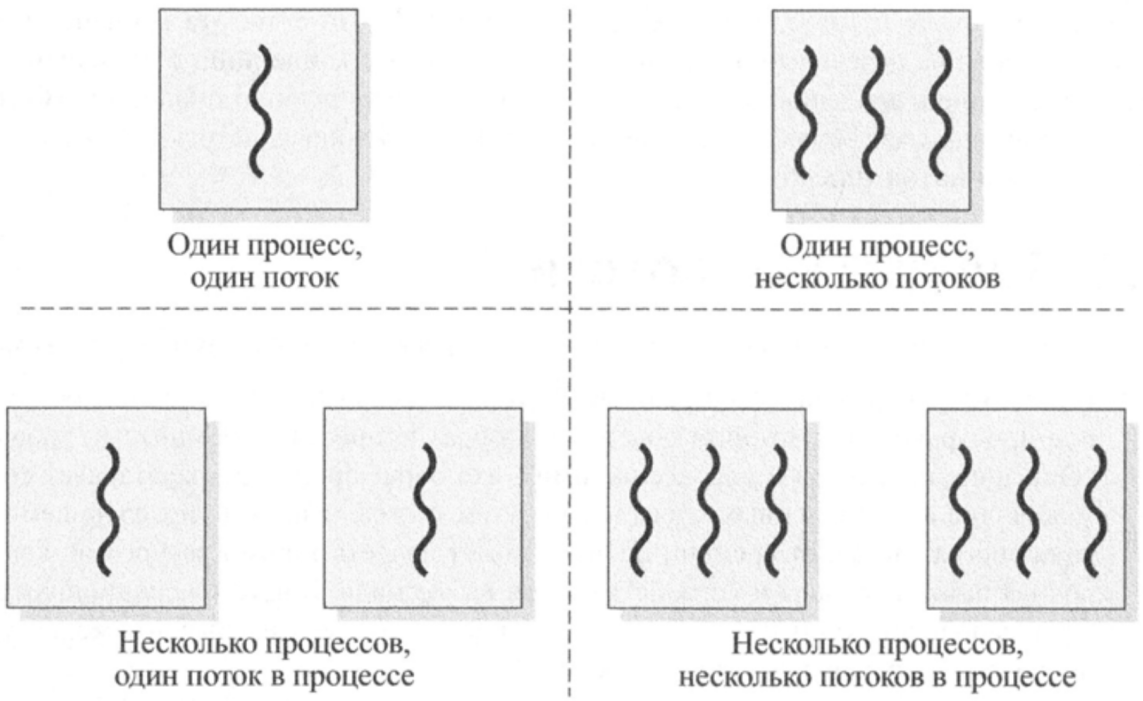
Процесс с точки зрения ОС:
1)владение процессом(ресурсы)
2)планирование/исполнение (как раз наши потоки)

Поток (нить выполнения, thread) - отдельный набор регистров и единица диспетчеризации и выполнения ОС (по сути это элемент выполнения процесса).

Многопоточностью (multithreading) называется способность операционной системы поддерживать в рамках одного процесса несколько параллельных путей выполнения.

Это удобно, потому что создавать весь процесс с нуля - дорого и долго.

Множественные нити исполнения в одном процессе называют *потоками* и это базовая единица загрузки CPU, состоящая из идентификатора потока, счетчика, регистров и стека.
Многопоточные модели:



Потоки POSIX

В конце 1980-х и начале 1990-х было несколько разных API, но в 1995 г. *POSIX.1c* стандартизировал потоки POSIX, позже это стало частью спецификаций *SUSv3*.

В программировании **зелёные потоки (англ. *green threads*)** — это потоки выполнения, управление которыми вместо операционной системы производит виртуальная машина (ВМ). Green threads эмулируют многопоточную среду, не полагаясь на возможности ОС по реализации легковесных потоков. Управление ими происходит в пользовательском пространстве, а не пространстве ядра, что позволяет им работать в условиях отсутствия поддержки встроенных потоков.

Преимущество заключается в том, что вы получаете функциональность, подобную обычному треду. Недостатком является то, что зеленые потоки фактически не могут использовать несколько ядер.

- Green threads значительно превосходят встроенные потоки Linux-системы по времени активации потоков и синхронизации.
- Встроенные потоки Linux имеют несколько более высокую производительность операций ввода-вывода и переключения контекста.

В Java 1.1 green threads являлись единственной потоковой моделью (моделью распараллеливания потоков), используемой в JVM, по крайней мере, в Solaris. Ввиду того, что green threads обладают ограничениями в сравнении с native threads, в последующих версиях Java основной упор сделан на native threads.

11. Симметричная и асимметричная многопроцессорная обработка.

- Asymmetric Multiprocessing — есть Master CPU, он управляет Slaves CPU
  - CPU — GPU
- Symmetric Multiprocessing – процессоры равны, процесс выполняется на нескольких процессорах одновременно
  - «Простота» разработки и производительность
  - Более высокая надежность. При отказе одного выполнять процессы могут другие
  - Масштабируемость приложений
  - Динамическое добавление ресурсов процессора
- Многопоточность ≠ Многопроцессорность

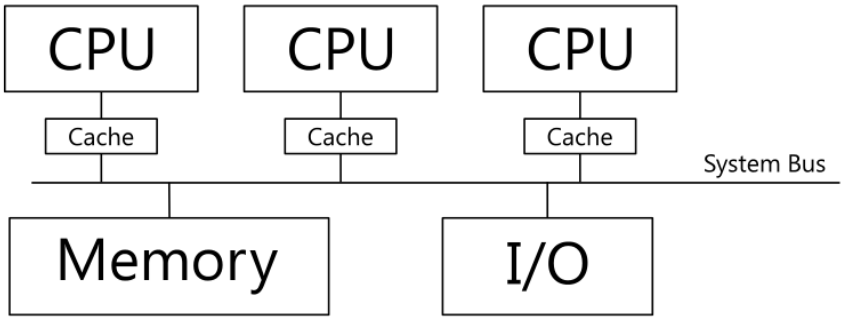
AMP или **ASMP** (от англ.: **Asymmetric multiprocessing**, рус.: **Асимметричная многопроцессорная обработка** или **Асимметричное мультипроцессирование**) — тип многопроцессорной архитектуры компьютерной системы, который использовался до того, как была создана технология симметричного мультипроцессирования (SMP). Также использовался как более дешевая альтернатива в системах, которые поддерживали SMP.

Асимметричное мультипроцессирование – наиболее простой способ организации вычислительного процесса в системах с несколькими процессорами. Этот способ часто называют также “ведущий-ведомый”.

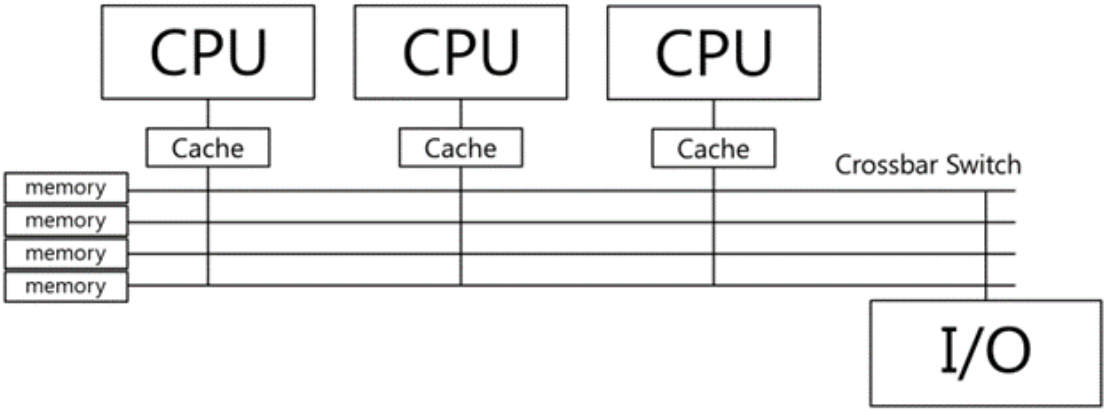
GPU – графический процессор (тут пример: что ЦПУ - главный и распределяет задачи, а ГПУ - просто процессит переданные задачи).

Симметричная многопроцессорность (англ. *symmetric multiprocessing*, сокращённо **SMP**) — архитектура многопроцессорных компьютеров, в которой два или более одинаковых процессора сравнимой производительности подключаются единообразно к общей памяти (и периферийным устройствам) и выполняют одни и те же функции (почему, собственно, система и называется *симметричной*). Просто, потому что все одинаково, но надо бороться со всякими локами и захватами одного и того же участка. Каждый процесс управляет собой.

Разные SMP-системы соединяют процессоры с общей памятью по-разному. Самый простой и дешевый подход — это соединение по **общей шине** (system bus/UMA). В этом случае только один процессор может обращаться к памяти в каждый данный момент, что накладывает существенное ограничение на количество процессоров, поддерживаемых в таких системах. Чем больше процессоров, тем больше нагрузка на общую шину, тем дольше должен ждать каждый процессор, пока освободится шина, чтобы обратиться к памяти. Снижение общей производительности такой системы с ростом количества процессоров происходит очень быстро, поэтому обычно в таких системах количество процессоров не превышает 2-4. Примером SMP-машин с таким способом соединения процессоров являются любые многопроцессорные серверы начального уровня.



Второй способ соединения процессоров — через **коммутируемое соединение** (crossbar switch/NUMA). При таком соединении вся общая память делится на банки памяти, каждый банк памяти имеет свою собственную шину, и процессоры соединены со всеми шинами, имея доступ по ним к любому из банков памяти. Такое соединение схемотехнически более сложное, но оно позволяет процессорам обращаться к общей памяти одновременно. Это позволяет увеличить количество процессоров в системе до 8-16 без заметного снижения общей производительности. Примером таких SMP-машин являются многопроцессорные рабочие станции RS/6000.



12. Виртуализация. Типы виртуализации.

Виртуализация — это создание виртуальной (логической) версии каких-то ресурсов на одной вычислительной машине . Т.е. это некий интерпретатор, работающий под управлением другой программы.

- Виртуальные машины (интерпретаторы)
  - Java VM, JavaScript в браузере, Python
- Контейнеры приложений
  - Docker, Solaris containers, Linux Containers (LXC), ...
- Аппаратная виртуализация
  - KVM, Hyper-V, VMWare, Virtual Box, ...
  - Виртуализация аппаратных устройств
- Облачные технологии
  - Построены на базе аппаратной виртуализации
  - Дополнительно включают provisioning и общий мониторинг

В целом, виртуальная машина – совокупность ресурсов, которые симулируют поведение реальной машины. При этом процессы идущие на host-машине и гостевой платформе изолированы.

Контейнеры приложений - поменять корень и чтобы все оттуда выполнялось, можно положить только нужные либы и ничего лишнего.
(Суть изоляции приложения заключается в том, чтобы оно «не знало» о других программах, работающих в той же операционной системе одновременно с ним, чтобы оно «считало» себя единственным работающим.) Пример: запуск в песочнице

Для каждого приложения во время его запуска создается, так называемая, среда выполнения.

Грубо говоря, виртуализация - эмуляция аппаратного окружения (несколько ОС на одной машине), контейнеризация - выделение изолированного окружения (контейнера) в рамках одной ОС. Все контейнеры используют одно ядро ОС, в виртуализации у каждого окружения свое ядро.

Облачные технологии позволяют использовать с устройства ресурсы удаленных устройств.

13. Сбои и отказоустойчивость ОС. Причины появления отказов в ОС и способы борьбы с ними..

- Способность системы продолжать работу при аппаратных или программных ошибках
  - Избыточность аппаратуры (двойное, тройное резервирование)
  - Аппаратная «горячая» замена компонентов (диски, контроллеры, процессоры, системные платы)
  - Программная поддержка ОС выведения компонентов из системы и их подключения
  - Организация уровней хранения RAID (Redundant Array of Inexpensive Disks) в дисковой подсистеме

Двойное резервирование: два устройства вместо одного (например, 2 драйвера). По дефолту можно размазать работу на них обоих, а если 1 выйдет из строя - ничего страшного, продолжаем работать: медленнее, но все же - система не рухнет.
Тройное резервирование: смотрится сигнал с всех 3х и берется “большинство” голосов.

Сложно ли менять в горячую? Надо уметь отключить банку (изолировать пассивно и активно), и затем ее заменить.

При наличии множества дисков различные запросы ввода-вывода могут обрабатываться параллельно, если блок данных, к которому производится обращение, распределен по множеству дисков. Кроме того, даже единственный запрос ввода-вывода может быть выполнен параллельно, если блок данных, к которому осуществляется доступ, распределен по нескольким дискам.

В случае применения нескольких дисков имеется большое количество вариантов организации данных и добавления избыточности для повышения надежности (что может

RAID - см. далее. Это дисковые массивы: дорого, зато эффективно(избыточный массив независимых дисков, рассматривается операционной системой как единый логический диск). Избыточность необходима для возможности восстановления данных в случае отказа одного из дисков (подробнее про уровни).

Таблица 11.4. Уровни RAID

Категория	Уровень	Описание	Требуемое количество дисков[1]	Доступность данных	Пропускная способность передачи больших данных	Скорость запросов малого ввода-вывода
Расщепление	0	Без избыточности	*N*	Меньше, чем у одного диска	Очень высокая	Очень высокая для чтения и записи
Отражение[2]	1	Отражение	2*N*	Больше, чем у RAID 2, 3, 4 и 5, но меньше, чем у RAID 6	Для чтения больше, чем у одного диска; для записи сравнима с одним диском	Для чтения почти вдвое больше, чем у одного диска; для записи сравнима с одним диском
Параллельный доступ	2	Избыточность с кодами Хэмминга	*N*+*m*	Гораздо выше, чем у одного диска, сравнимо с RAID 3, 4 и 5	Наибольшая среди всех перечисленных альтернатив	Почти вдвое больше, чем у одного диска
	3	Четность с чередующимися битами	*N*+1	Гораздо выше, чем у одного диска, сравнимо с RAID 2, 4 и 5	Наибольшая среди всех перечисленных альтернатив	Почти вдвое больше, чем у одного диска
Независимый доступ	4	Четность с чередующимися блоками	*N*+1	Гораздо выше, чем у одного диска, сравнимо с RAID 2, 3 и 5	Для чтения аналогична RAID 0; для записи значительно меньше, чем у одного диска	Для чтения аналогична RAID 0; для записи значительно меньше, чем у одного диска
	5	Распределенная четность с чередующимися блоками	*N*+1	Гораздо выше, чем у одного диска, сравнимо с RAID 2, 3 и 4	Для чтения аналогична RAID 0; для записи меньше, чем у одного диска	Для чтения аналогична RAID 0; для записи в общем случае меньше, чем у одного диска
	6	Двойная распределенная четность с чередующимися блоками	*N*+2	Наибольшая среди всех перечисленных альтернатив	Для чтения аналогична RAID 0; для записи меньше, чем у RAID 5	Для чтения аналогична RAID 0; для записи значительно меньше, чем у RAID 5

====

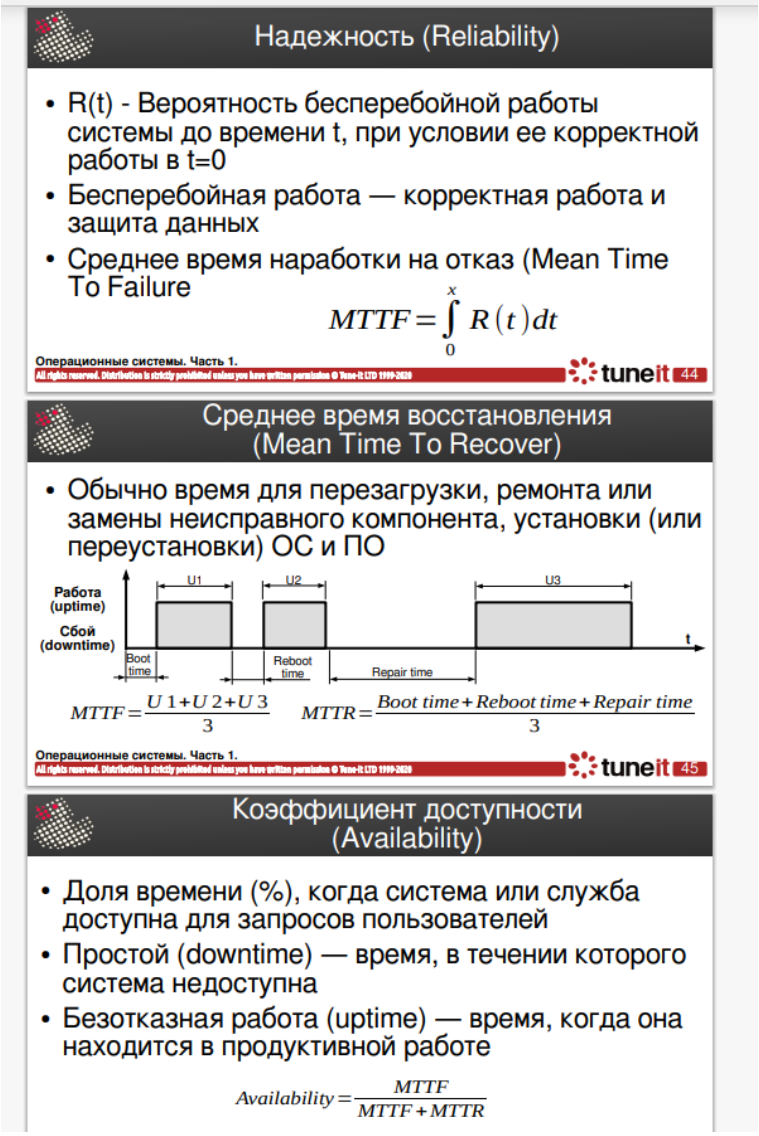
ВСЕ эти моменты должны быть решены самой ОС.

Причины отказов:

- Ошибочное состояние аппаратуры или ПО в результате сбоя компонентов (может и из-за лунных волн, а может постоянно, тогда уже программа в аппаратуре)
- Ошибки оператора (ввел что-то не так:)
- Физические помехи окружающей среды (кабель влияет на кабель)
- Ошибки проектирования, программирования, структур данных и пр.

Могут быть: постоянные, временные (однократные или периодические)

14. Надежность. Среднее время восстановления. Коэффициент доступности и время простоя.



Надежность (reliability) $R(t)$ системы определяется как вероятность ее бессбойной работы до времени t при условии ее корректной работы в момент времени $t = 0$. Для операционных систем и компьютеров термин *бессбойная работа* означает правильное выполнение набора программ и защиту данных от случайного изменения. **Среднее время наработки на отказ** (mean time to failure — MTTF) определяется как

$$\text{MTTF} = \int_0^x R(t)dt$$

MTTR – среднее время восстановления

Доступность = вероятность, что система находится в рабочем состоянии

Ну тут вроде все понятно.
Есть еще классы доступности систем (процент коэффа доступности), но это +_ условная штука.

15. Резервирование и отказоустойчивость.

Методы резервирования

- Физическая избыточность (компонентов, серверов) - использовать два прибора/контроллера для одного и того же.
- Временная избыточность (повтор вычислений)
- Информационная избыточность (ECC, RAID - контрольные суммы)

Отказоустойчивость означает способность системы или компонента к продолжению нормальной работы, несмотря на наличие ошибок аппаратного или программного обеспечения. Обычно отказоустойчивость предполагает определенную степень избыточности. Отказоустойчивость предназначена для повышения степени надежности системы.Как правило, увеличение отказоустойчивости (и соответственно, повышение надежности) имеет определенную стоимость, либо финансовую, либо выражающуюся в падении производительности (либо и то, и другое одновременно). Таким образом, определение желаемой степени отказоустойчивости должно учитывать, что именно является критическим ресурсом.

Методы повышения отказоустойчивости ОС

- Изоляция процессов (сейчас с виртуальным пространством очень удобно, но в ядре все еще одно пространство).
- Разрешение блокировок при параллелизме.
- Виртуализация - полностью изолировать выполнение (свои ядра).
- Точки восстановления (копии ключевых файлов) и откаты.

16. История и развитие ОС GNU/Linux. Single UNIX Specification и POSIX.

История

Система Linux возникла как вариант операционной системы UNIX, предназначенный для персональных компьютеров с архитектурой IBM PC (Intel 80386). Первоначальная версия была написана Линусом Торвальдсом (Linus Torvalds), финским студентом, изучавшим теорию вычислительных машин. В 1991 году Торвальдс представил в Интернете первую версию системы Linux. С тех пор множество людей, сотрудничая посредством Интернета, развивают Linux под общим руководством ее создателя. Благодаря тому что система Linux является бесплатной и можно беспрепятственно получить ее исходный код, она стала первой альтернативой для рабочих станций UNIX, предлагавшихся фирмами Sun Microsystems и IBM. На сегодняшний день Linux является полнофункциональной системой семейства UNIX, способной работать почти на всех платформах.

Залогом успеха Linux является то, что она бесплатно распространяется при поддержке Фонда бесплатно распространяемых программ (Free Software Foundation — FSF). Целью этой организации является создание надежного аппаратно-независимого программного обеспечения, которое было бы бесплатным, обладало высоким качеством и пользовалось широкой популярностью среди пользователей. Проект GNU[2] фонда предоставляет инструменты для разработки программного обеспечения под эгидой общедоступной лицензии GNU (GNU Public License — GPL). Таким образом, система Linux в таком виде, в котором она существует сегодня, является продуктом, появившимся в результате усилий Торвальдса, а затем и многих других его единомышленников во всем мире, и распространяющимся в рамках проекта GNU.

Linux используется не только многими отдельными программистами; она проникла и в корпоративную среду. В основном это произошло благодаря высокому качеству ядра операционной системы Linux, а не из-за того, что эта система является бесплатной. В эту популярную версию внесли свой вклад многие талантливые программисты, в результате чего появился впечатляющий технический продукт. К достоинствам Linux можно отнести то, что она является модульной и легко настраиваемой. Благодаря этому можно достичь высокой производительности ее работы на самых разнообразных аппаратных платформах. К тому же, получая в свое распоряжение исходный код, производители программного обеспечения могут улучшать качество приложений и служебных программ, с тем чтобы они удовлетворяли конкретным требованиям их пользователей. Имеются также коммерческие компании, такие как Red Hat и Canonical, которые обеспечивают высокопрофессиональную и надежную поддержку своих дистрибутивов Linux. В этой книге подробности внутреннего устройства ядра Linux излагаются на основе ядра Linux 4.7, выпущенного в 2016 году.

Большая часть успеха операционной системы Linux связана с используемой ею моделью развития. Разработчики пользуются единым списком рассылки под названием "LKML" (Linux Kernel Mailing List — список рассылки ядра Linux). Кроме того, имеется множество других списков рассылки, каждый из которых посвящен той или иной подсистеме ядра Linux (список рассылки netdev для сети, linux-pci — для подсистемы PCI, linux-acpi — для подсистемы ACPI и др.). Обновления, отправляемые в эти списки рассылки, должны соответствовать строгим правилам (главным образом — соглашениям о кодировании ядра Linux) и изучаются разработчиками со всего мира, подписанными на эти списки рассылки. Любой пользователь может отправлять свои обновления в эти списки рассылки. Статистика (например, время от времени публикуемая на сайте `lwn.net`) показывает, что многие обновления предлагаются разработчиками из известных коммерческих компаний, таких как Intel, Red Hat, Google, Samsung и др. Кроме того, многие разработчики являются сотрудниками коммерческих компаний (как Дэвид

того, многие разработчики являются сотрудниками коммерческих компаний (как Дэвид Миллер (David Miller), поддерживающий сетевые функции и работающий в компании Red Hat). Такие обновления изучаются и обсуждаются в списке рассылки, после чего в них вносятся исправления, и цикл обсуждения начинается заново. В конце концов принимается решение, следует ли принять или отклонить эти исправления. Каждый руководитель подсистемы время от времени отправляет запрос о размещении исправлений его части в основном ядре, который обрабатывается Линусом Торвальдсом. Сам Линус

[2] GNU — рекурсивная аббревиатура для GNU's Not Unix (GNU — не Unix). Проект GNU представляет собой бесплатный набор программных пакетов и инструментов для разработки UNIX-подобной операционной системы.

выпускает новую версию ядра примерно каждые 7–10 недель, причем каждый такой выпуск имеет около 5–8 предварительных версий-кандидатов.

Интересно попытаться понять, почему другие операционные системы с открытым кодом, такие как различные версии BSD или OpenSolaris, не имеют таких успеха и популярности, которыми обладает Linux. Тому может быть много причин; конечно, открытость модели развития Linux способствовала популярности и успеху этой операционной системы. Но эта тема выходит за рамки нашей книги.

Single UNIX Specification. POSIX

- Unix SUS: The Open Group, Austin Group
  - Основные определения
  - Системные интерфейсы
  - Командная оболочка и утилиты
  - Пояснения
  - X/Open Curses
  - UNIX: AIX, HP-UX, IRIX, Mac OS X, SCO OpenServer, Solaris, Tru64 и z/OS.
  - UNIX-like: FreeBSD, OpenBSD, NetBSD, OpenSolaris, BeleniX, Nexenta OS) и Linux
- POSIX (Portable Operating System Interface) ISO/IEC 9945

SUS – спецификация, предъявляющая требования к UNIX системам. При помощи данного документа можно определить, относится ли ОС к Unix
POSIX - стандарт отвечает за интерфейс ОС UNIX
(ISA - instructions set architecture, ABI - application binary interface, API - application program interface).

UNIX-like – операционки, не удовлетворяющие стандартам

17. Понятие дистрибутива, дистрибутивы Linux.

Дистрибутив - это ядро + набор софта для прикладных задач, этот набор в каждом дистрибутиве может быть свой. В этот набор входит “системный” софт: пакетный

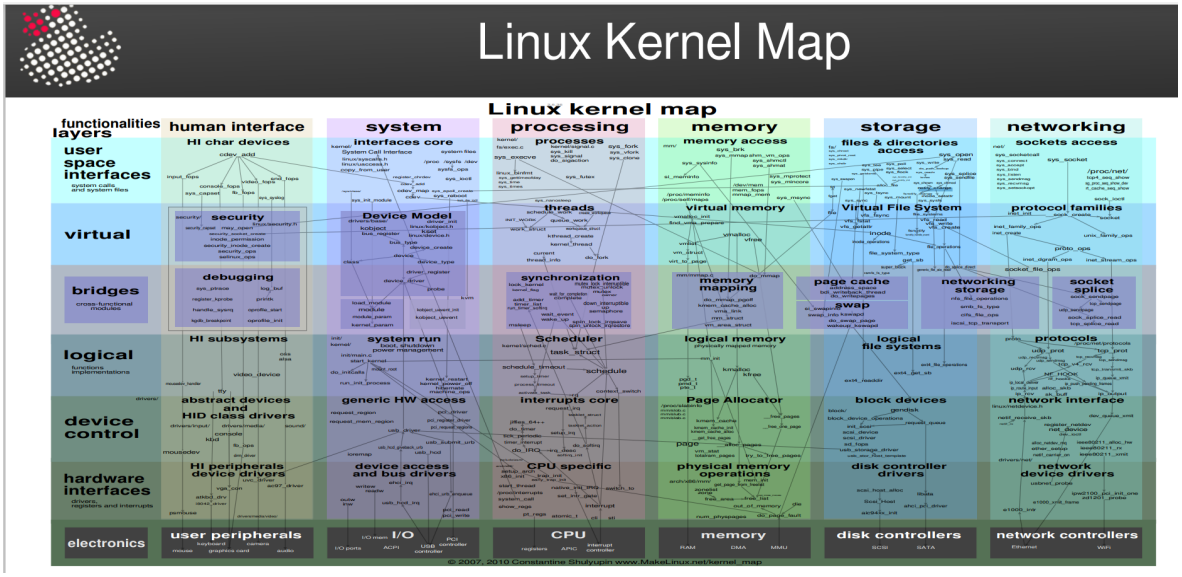
менеджер, файловый менеджер, графическая оболочка и набор прикладных программ (текстовый редактор, текстовый процессор, браузер, аудио/видеопроигрыватели и т.п.) (но при этом далеко не все это обязательно входит, например, графическая оболочка далеко не везде используется, а некоторые дистрибутивы представляют из себя почти чистое ядро, например Arch)
Часть дистрибутивов создается компаниями(RedHat, Ubuntu, Suse),: хотя это и опен-сурс проекты, компании продают дистрибутивы, предлагая поддержку (обновления, траблшутинг)
Многие дистрибутивы поддерживаются сообществом, которое использует их и заинтересовано в их развитии (Arch, Debian и их многочисленные потомки)
Поскольку код открытый, многие берут старый дистрибутив за основу и что-то меняют там, создавая новый - этим объясняется огромное дерево дистрибутивов Линукса, но большинство из них наследники Debian и RedHat. Android использует ядро Linux, но не GNU, поэтому обычно не причисляется к дистрибутивам Linux.

В Linux ядро обычно берется от Линуса Торвальдса(какой-либо версии) и включает в себя окружение(библиотеки, утилиты). Дистрибутивы отличаются различными менеджерами пакетов и обновлений. Кроме того есть графическая подсистема, есть какие-то прикладные программы, которые работают/не работают :) на Linux. Поддержку дистрибутива можно купить за денежку.

18. Архитектура и основные подсистемы Linux. Linux Kernel Map.

Архитектура Linux

Рис 1 Компоненты Linux



https://makelinux.github.io/kernel/map/

Основные подсистемы Unix/Linux:

- Процессы и планировщик. – Создает, управляет и планирует процессы.
- Виртуальная память. – Выделяет виртуальную память для процессов и управляет ею.
- Физическая память. – Управляет пулом кадров страниц и выделяет страницы для виртуальной памяти.

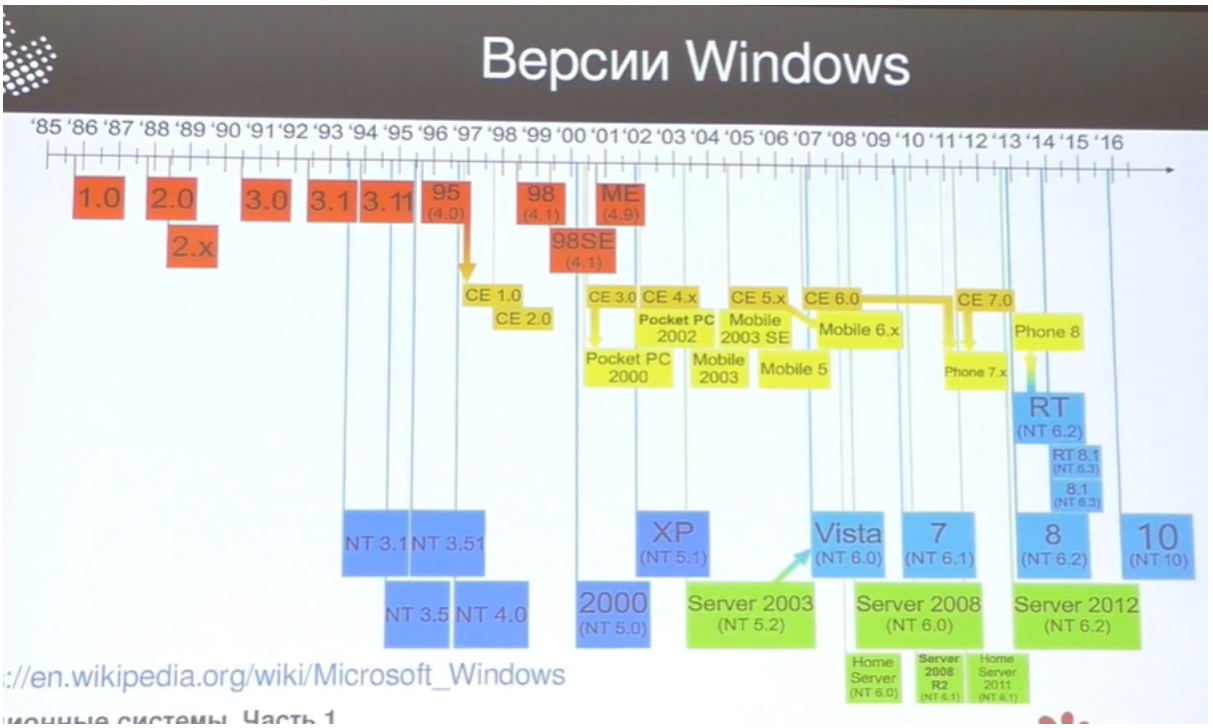
(Физическая и виртуальная память связаны через MMU manager memory unit, много вкручено функций на уровне ядра, управляющих этим взаимодействием).

- Файловая система (бывают физические и виртуальные) – Предоставляет глобальное иерархическое пространство имен для файлов, каталогов и других объектов, связанных с файлами и функциями файловой системы.
  В Unix все есть файл, поэтому все, с чем работаем - файловая система. Любая система обращается к драйверу vfs (virtual file system).
- Драйверы символьных устройств. (непосредственно передают данные на устройство) – Управление устройствами, которые требуют от ядра отправки или получения данных по одному байту, например терминалами, принтерами или модемами.
  (ls -l: chain-oriented)
- Драйверы блочных устройств. (осуществляют буферизацию сначала) – Управление устройствами, которые читают и записывают данные блоками, как, например, различные виды вторичной памяти (магнитные диски, CD-ROM и т.п.).
  (ls -l: block-oriented)
- Сетевые протоколы. TCP/IP. – Поддержка пользовательского интерфейса сокетов для набора протоколов (еще, например x25 - в банковской сфере). Их оч. много.
- Драйверы сетевых устройств. – Управление картами сетевых интерфейсов и коммуникационными портами, которые подключаются к сетевым устройствам, такими как мосты или роутеры.

- Ловушки и отказы. – Обработка генерируемых процессором прерываний, как, например, при сбое памяти. (+ реакция: паника, подгрузка страничек…)
- Прерывания. – Обработка прерываний от периферийных устройств.
- Сигналы и IPC – Управляет межпроцессным взаимодействием.

Утилита для посылки сигналов: kill, какие есть сигналы: kill -l.
IPC - средство межпроц. взаимодействия. Включает: разделяемая память, семафоры, месседжи.

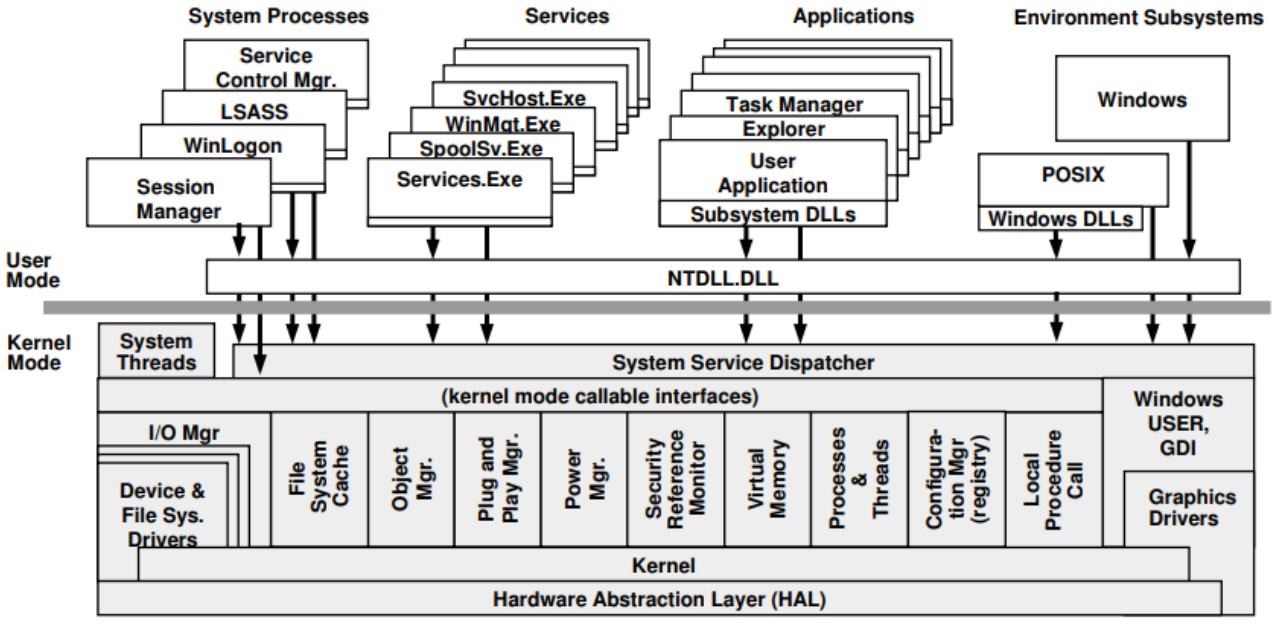
19. История и развитие Windows

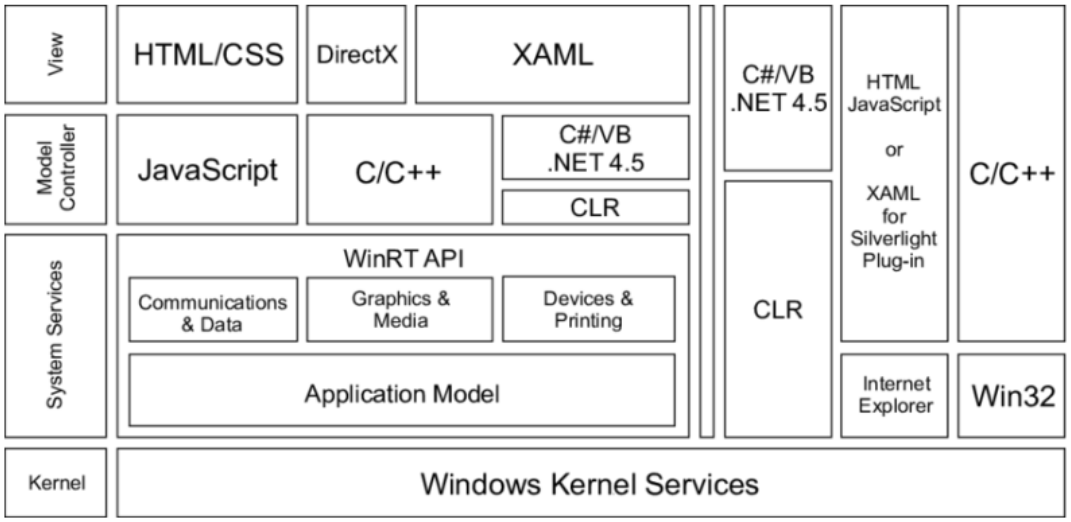


Основы

Впервые Microsoft использовала имя "Windows" в 1985 году для операционной среды, расширяющей возможности примитивной операционной системы MS-DOS, которая успешно использовалась на ранних персональных компьютерах. Такая комбинация Windows/MS-DOS в конечном итоге была заменена новой версией Windows, известной как Windows NT, впервые выпущенной в 1993 году и предназначенной для ноутбуков и настольных систем. Хотя основная внутренняя архитектура остается примерно той же, что и в Windows NT, сама операционная система продолжает развиваться и дополняться новыми функциями и возможностями. Последняя версия на момент написания этой книги — Windows 10. Windows 10 включает в себя возможности предыдущей операционной системы для настольных и портативных компьютеров Windows 8.1, а также версий Windows, предназначенных для мобильных устройств для Интернета вещей (Internet of Things — IoT). Windows 10 также включает в себя программное обеспечение Xbox One. В результате единая унифицированная операционная система Windows 10 поддерживает настольные компьютеры, ноутбуки, смартфоны, планшеты и Xbox One.

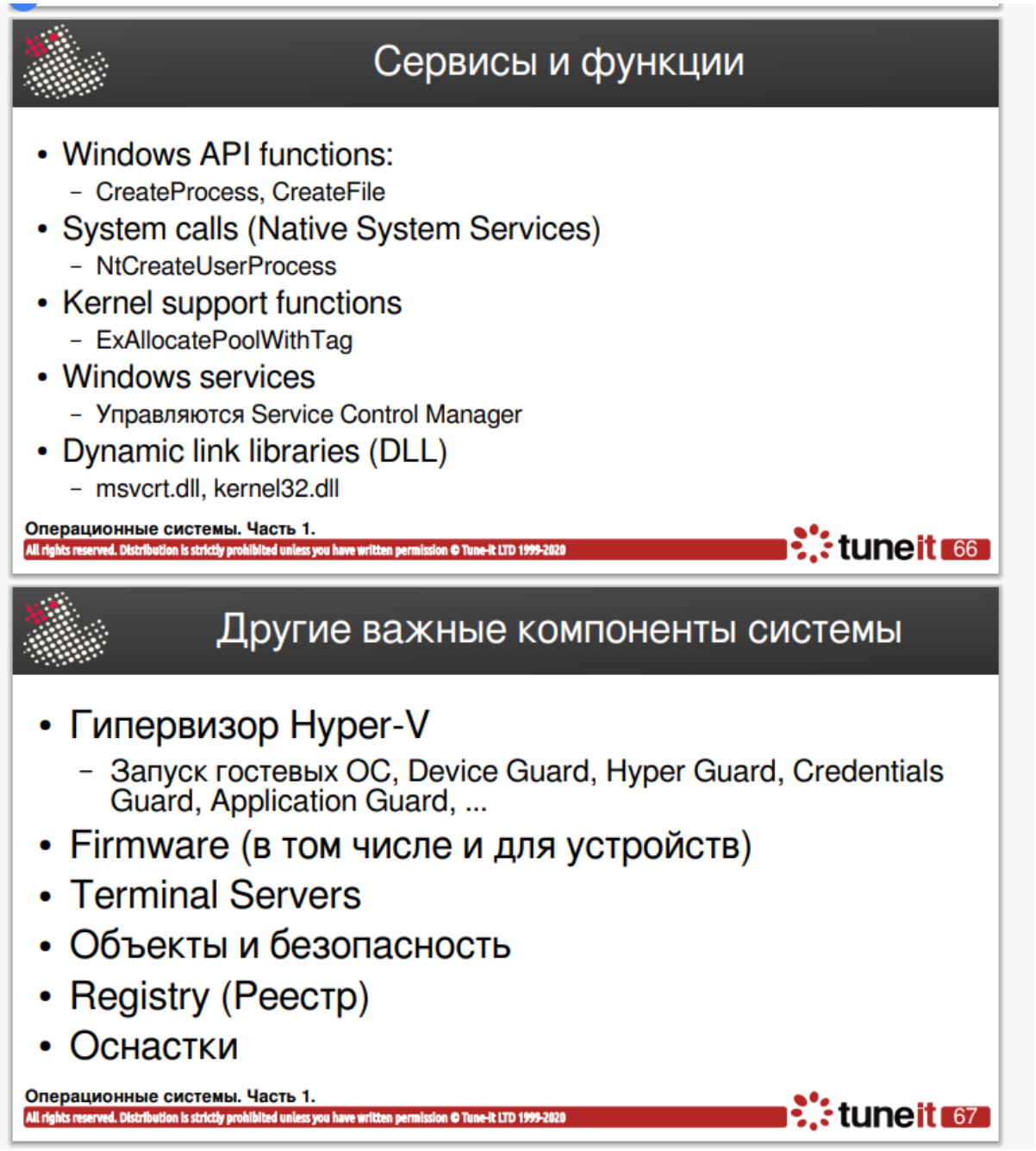
20. Общая архитектура Windows. Windows API





Windows присуще четкое разделение на модули. Каждая функция системы управляется только одним компонентом операционной системы. Остальные ее части и все приложения обращаются к этой функции через стандартный интерфейс. Доступ к основным системным данным можно получить только через определенные функции. В принципе любой модуль можно удалить, обновить или заменить, не переписывая всю систему или стандартный интерфейс прикладного программирования (application program interface -API).

21. Сервисы, функции и важные компоненты Windows.



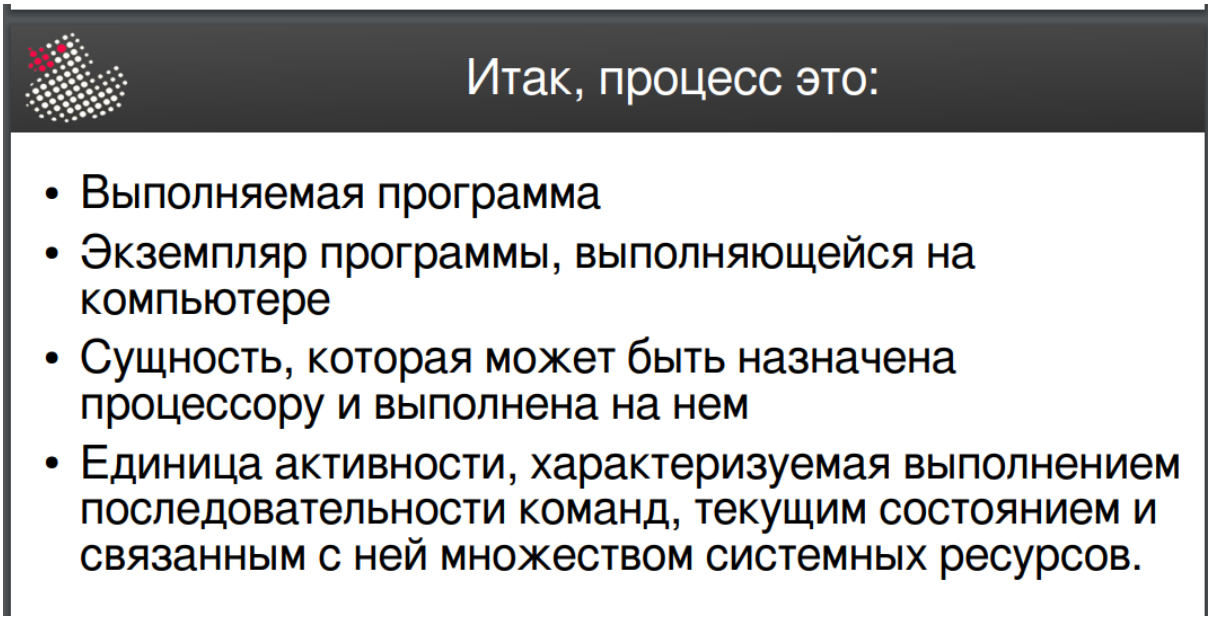
Сервисы и функции

- Windows API functions:
  - CreateProcess, CreateFile
- System calls (Native System Services)
  - NtCreateUserProcess
- Kernel support functions
  - ExAllocatePoolWithTag
- Windows services
  - Управляются Service Control Manager
- Dynamic link libraries (DLL)
  - msvcrt.dll, kernel32.dll

Другие важные компоненты системы

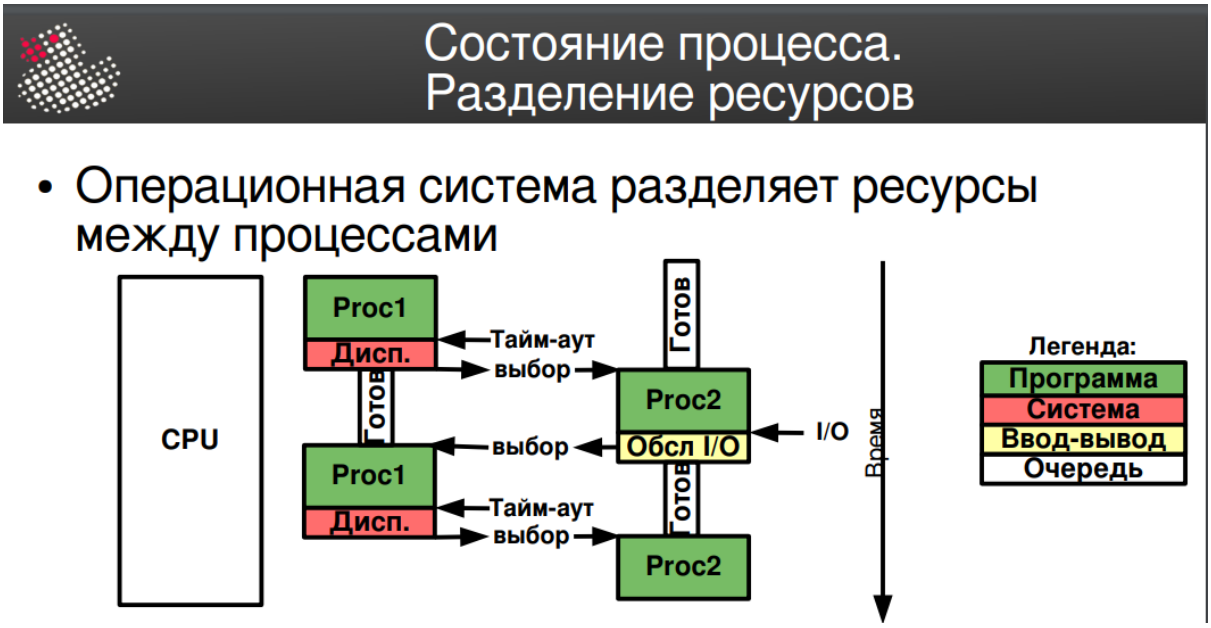
- Гипервизор Hyper-V
  - Запуск гостевых ОС, Device Guard, Hyper Guard, Credentials Guard, Application Guard, ...
- Firmware (в том числе и для устройств)
- Terminal Servers
- Объекты и безопасность
- Registry (Реестр)
- Оснастки

22. Процесс, характеристики процесса в момент выполнения. Состояние процесса. Разделение ресурсов.



- **Идентификатор**. Уникальный идентификатор, связанный с этим процессом, чтобы отличать его от всех прочих процессов.
- **Состояние**. Если процесс выполняется в настоящее время, он находится в **состоянии выполнения**.
- **Приоритет**. Уровень приоритета по отношению к другим процессам.
- **Программный счетчик**. Адрес очередной выполняемой команды программы.
- **Указатели памяти**. Включают указатели на программный код и данные, связанные с этим процессом, а также на любые блоки памяти, совместно используемые с другими процессами.
- **Данные контекста**. Это данные, присутствующие в регистрах процессора во время выполнения процесса.
- **Информация о состоянии ввода-вывода**. Включает в себя внешние запросы ввода-вывода, устройства ввода-вывода, назначенные процессу, список файлов, используемых процессом, и т.д.
- **Учетная информация**. Может включать количество процессорного времени и времени работы, учетные записи и т.д.

Рис. 3.1. Упрощенный управляющий блок процесса



Эту схему я честно говоря не особо понял, но судя по всему тут имеется ввиду многозадачность, что ОС распределяет процессорное время между разными процессами и у этих процессов разные состояния. 1 процесс вначале выполняется, затем завершается по тайм ауту и становится готовым к выполнению, в то время как 2 процесс начинается выполняться и заканчивает свое выполнение по прерыванию от ввода/вывода, после чего становится блокированным и так далее.

Здесь речь о системе разделения ресурсов: у нас есть очередь процессов и процессам отведено равное время на исполнение, как оно истекает, происходит переход к следующему процессу(в состоянии готовности), который исполнялся наименьшее время. Хотим таким образом достичь “честного” распределения ресурсов и не допустить starvation.

23. Модель процесса с пятью состояниями, назначение состояний.

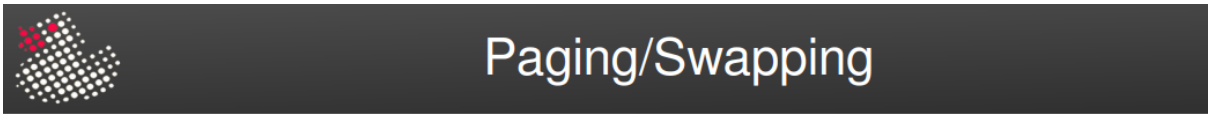
Рис. 3.6. Модель с пятью состояниями

Опишем каждое из пяти состояний процессов, представленных на диаграмме.

1. **Выполняющийся**. Процесс, который выполняется в текущий момент времени. В настоящей главе предполагается, что на компьютере установлен только один процессор, поэтому в этом состоянии может находиться только один процесс.
2. **Готовый к выполнению**. Процесс, который может быть запущен, как только для этого представится возможность.
3. **Блокированный/Ожидающий**[5]. Процесс, который не может выполняться до тех пор, пока не произойдет некоторое событие, например завершение операции ввода-вывода.

4. **Новый**. Только что созданный процесс, который еще не помещен операционной системой в пул выполнимых процессов. Обычно это новый процесс, который еще не загружен в основную память, хотя управляющий блок процесса уже создан.
5. **Завершающийся**. Процесс, удаленный операционной системой из пула выполнимых процессов из-за завершения его работы или аварийно прерванный по какой-либо иной причине.

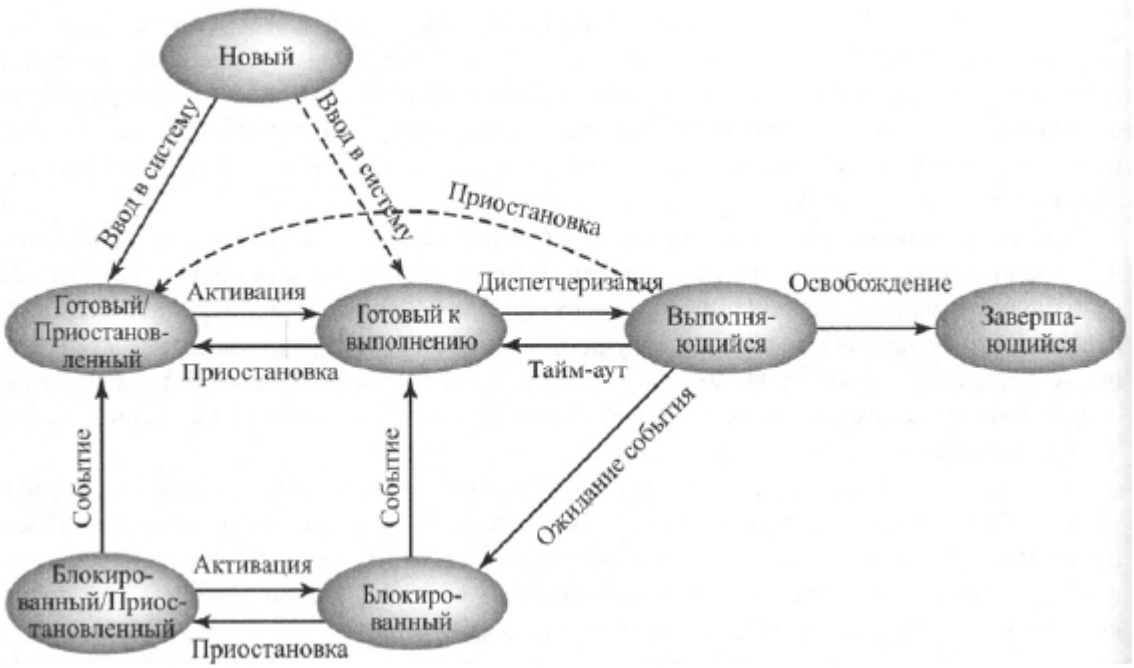
24. Paging и Swapping. Модель процесса с семью состояниями.



- Основной памяти всегда мало =(
  - Программисты как только видят больше памяти, стараются ее максимально «использовать»
  - Большое количество запущенных процессов
- Давайте поместим блокированный процесс на диск и освободим основную память для других процессов!
  - Нужно организовать область подкачки процессов на диске
- Paging (пейджиг) — выгрузка (и загрузка) неиспользуемых страниц процесс на диск
- Swapping (свопинг) — выгрузка всего процесса, кроме критически важных для ядра структур управления

В принципе, Paging – это тот же Swapping, но осуществляемый «постранично»: если вдруг оказывается, что какая-то страничка не очень нужна, то она выгружается. Для этого в операционных системах существуют специальные алгоритмы, занимающиеся вытеснением из памяти не использующихся страниц. Такой алгоритм сводится к тому, что у страниц, находящихся в памяти «по кругу» постоянно проверяется – есть ли изменения информации на странице. И если изменений на странице нет в течение заданного времени, то страница считается неиспользуемой и специальным образом помечается. А далее, в зависимости от того, что это за страница, принимается решение, что с ней делать. Например, если на странице находится код программы, который всегда можно загрузить из файла заново, то эту страницу можно смело удалить; если же на странице находятся данные, то ее необходимо сохранить в своп, и так далее.

Модель процесса с 7 состояниями



Следует обратить внимание на то, что на схеме появились новые состояния процесса.

Одно из них - это состояние runnable/suspend, при котором процесс может выполнится, но - либо весь процесс, либо его существенная часть находится в области подкачки.

Для простоты дальнейшего описания сейчас будем говорить о Swapping, учитывая то, что при Paging все очень похоже.

Итак, процесс готов на выполнение, у него ввод-вывод закончился. И процесс хочет попась на процессор. Но ему мешает то, что его часть находится в свопе. А оказалась она там потому, что до этого у процесса было другое состояние wait(suspend). Рассмотрим подробнее.

Вначале процесс находился на CPU. Потом процесс «перешел к вводу-выводу» или «повис» на какой-либо блокировке. В какой-то момент операционная система «решила», что процесс блокировки тянется слишком долго (или ОС решила, что памяти слишком мало), поэтому операционная система решает освободить от этого процесса память, потому что он все равно ничего не делает и находится на блокировке. После этого процесс переходит в состояние wait(suspend). После этого в какой-то момент «ввод-вывод» «пришел», и процесс необходимо «пробудить». Система «перебрасывает» процесс в очередь ожидания запуска. Но процесс выполнится не может, потому что его часть находится на диске. После этого запускается процедура, загружающая процесс с диска в память. И только после того, как все необходимые страницы загружены, процесс переходит в состояние runnable и может быть выполнен на процессоре.

В этой схеме есть еще ряд интересных вещей. На схеме явно видно, что мы разделили состояние wait на две части. И так же разделили на две части состояние runnable. Помимо этого, на схеме появились «стрелочки», которые могут вызывать вопросы. Дело в том, что процесс порождения процесса очень длителен и он занимает сотни миллисекунд (а иногда и больше). И все современные системы, учитывая наличие виртуальной памяти и наличие страничек, поступают следующим образом. Чтобы процесс создать как можно быстрее, он создается практически без создания маппингов памяти. И может получиться так, что при приходе процесса в состояние runnable/ desperate memory conditions, в его сегменте кода не будет ни одной загруженной страницы. И поэтому процесс не сможет продолжить свое выполнение. При этом получается, что процесс уже создан, все структуры ядра созданы, но процесс выполнится еще не может. Поэтому необходимо задействовать уже механизмы пейджинга и свопинга для подкачки необходимых для выполнения страниц.

Также у операционной системы в зависимости от количества свободной памяти есть несколько режимов работы. И на одной из вновь появившихся стрелочек происходит desperate memory conditions – это состояние системы, когда процессы потребляют столько памяти, что ее начинает не хватать самой системе. В этот момент операционная система начинает массово отправлять процессы в своп, чтобы освободить память для своих нужд. Таким образом процесс оказывается полностью готов, но у него нет возможности выполнится, потому, что у операционной системы не хватит ресурсов, чтобы его выполнить. И очень часто, когда система очень загружена, случается так, что все процессы встают в «локи» на моменте выделения новой памяти.

25. Управляющие таблицы процесса. Образ процесса

Если смотреть на операционную систему укрупненно, то у нее есть четыре основных подсистемы ядра и некоторые внутренние структуры, которые их описывают.

В частности, существует подсистема с ее собственными внутренними структурами; существуют файлы и структуры файловой системы, обеспечивающие работу с файлами. Аналогичными структурами подсистем можно описать устройства и процессы.

Внутри операционной системы существуют списки процессов. Примерами таких списков могут являться «глобальный список» процессов и «список состояний», в которых процессы могут находиться.

У каждого процесса есть структура, которая отражает образ процесса в памяти машины. В эту структуру обычно входит «ядерная часть» и «часть на стороне пользователя», которая описывает и характеризует процесс, а также содержит данные процесса и некоторые другие характеристики. Таких структур внутри ядра большое количество, и они между собой сильно «перевязаны». При этом каждый разработчик драйвера создает, как правило, свои управляющие структуры.

«На стороне ядра» всегда должна находиться информация, которая показывает, что происходит с процессом, в каком он находится состоянии, что ему требуется для исполнения, сколько времени он потратил на CPU...

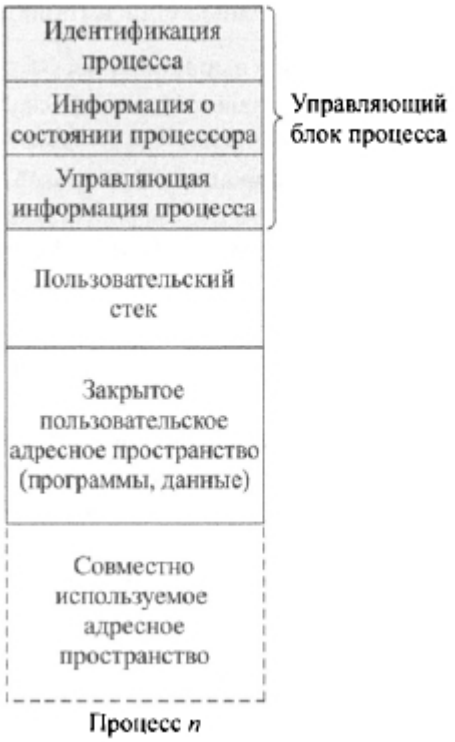
Если посмотреть на реальную структуру операционной системы (например, в Linux или в Solaris), можно заметить большое количество полей, для понимания которых необходимо потратить достаточное количество времени. Они сложны по названиям, и перемещаться по их перекрестным ссылкам достаточно сложно, так как часто одни структуры «вкладываются» в другие структуры, что в результате приводит к очень объемным текстам Си-шного кода, которые трудны для понимания.

Детального понимания всех этих структур для каждодневной работы «обычным программистам», как правило, не требуется. Однако для формирования качественного представления о предмете, имеет смысл посмотреть, как в операционной системе реализованы основные структуры.

Рис. 3.11. Общая структура управляющих таблиц операционной системы

Таблица 3.4. Типичные элементы образа процесса

Данные пользователя	Допускающая изменения часть пользовательского адресного пространства. Сюда могут входить данные программы, пользовательский стек и модифицируемый код
Пользовательская программа	Программа, которую нужно выполнить
Системный стек	С каждым процессом связан один или несколько системных стеков. Стек используется для хранения параметров, адресов вызова процедур и системных служб
Управляющий блок процесса	Данные, необходимые операционной системе для управления процессом (см. табл. 3.5)

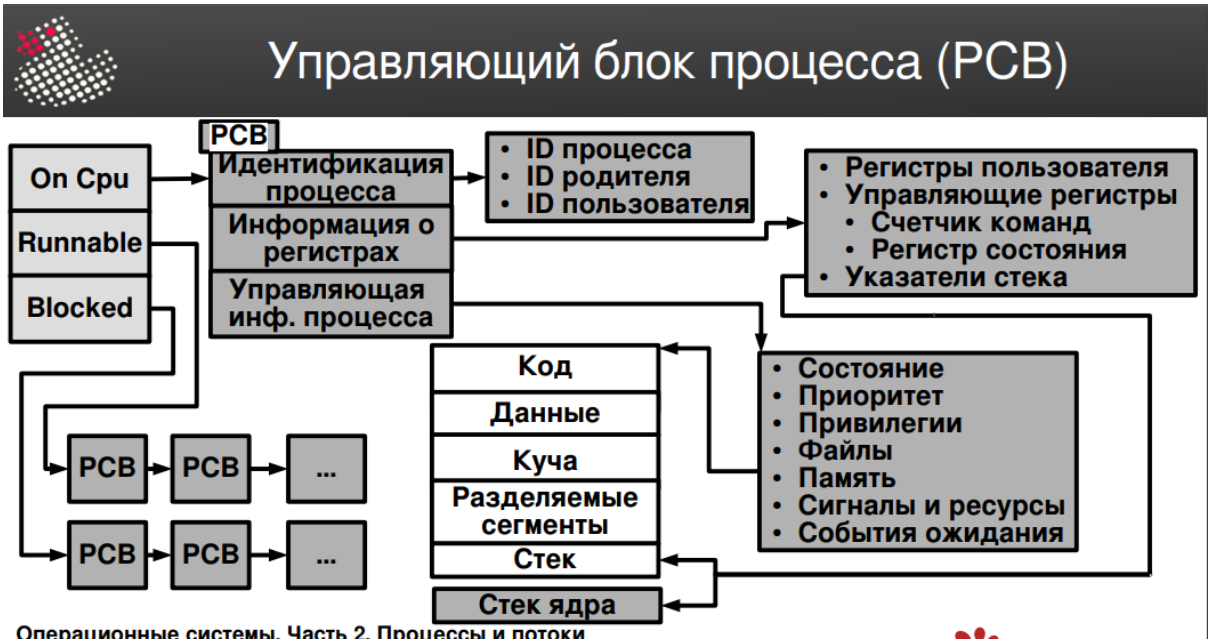


26. Управляющий блок процесса (PCB), состав PCB.

Роль управляющего блока процесса

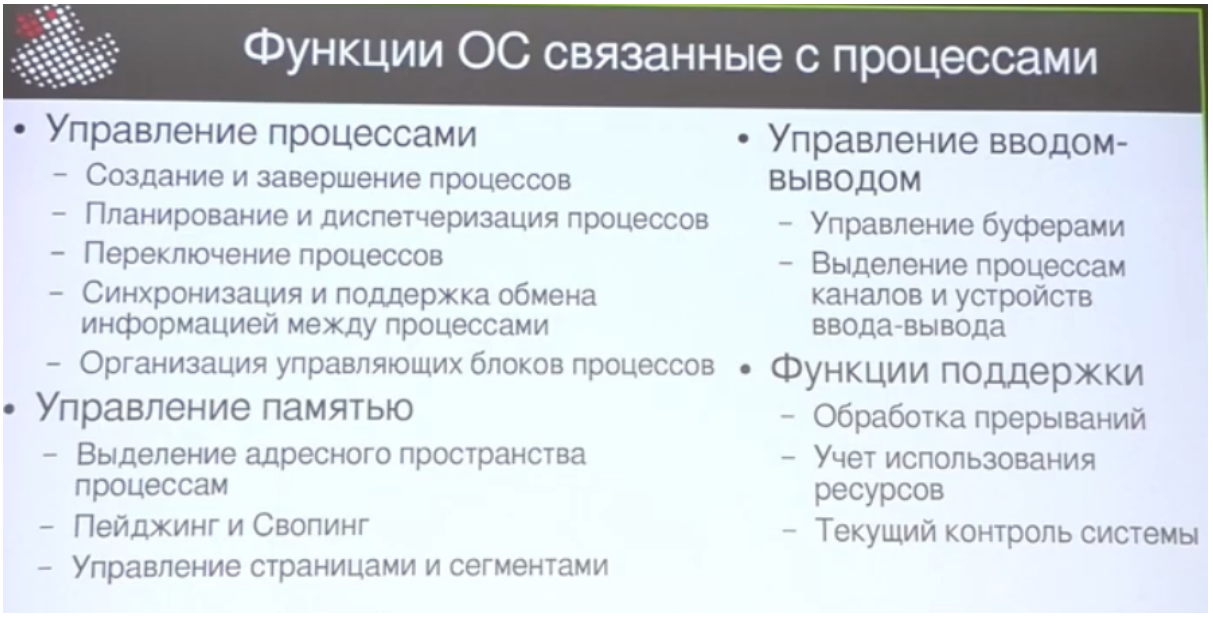
Управляющий блок процесса — это самая важная структура данных из всех имеющихся в операционной системе. В управляющий блок каждого процесса входит вся необходимая операционной системе информация о нем. Информация в этих блоках считывается и/или модифицируется почти каждым модулем операционной системы, включая те, которые связаны с планированием, распределением ресурсов, обработкой прерываний, а также осуществлением контроля и анализа. Можно сказать, что состояние операционной системы задается совокупностью управляющих блоков процессов.

Ю



Те, как мы видим в PCB содержится вся информация о процессе, его айди, состояние регистров, код, данные и тп.

27. Функции ОС, связанные с процессами. Создание процесса, переключение процессов.



Создание процесса

- Присвоить процессу уникальный идентификатор
- Выделить память для процесса
- Инициализировать PCB
- Поставить процесс в очереди ядра
- Создать потоки ввода-вывода
- Создать другие управляющие структуры данных

Переключение процессов

- Процесс может работать в user и kernel mode
- Используется механизм прерываний:
  - Внешнее прерывание (IO)
  - Ловушка — обработка ошибки или исключительной ситуации
  - Вызов ОС
- Переход из user в kernel-mode ситуации:
  - Прерывания таймера
  - Прерывания ввода-вывода
  - Page fault — отстутствие блока памяти

Таблица 3.8. Механизмы прерывания выполнения процесса

Механизм	Причина	Что используется
Прерывание	Внешняя по отношению к выполнению текущей команды	Отклик на внешнее асинхронное событие
Ловушка	Связана с выполнением текущей команды	Обработку ошибки или исключительной ситуации
Вызов супервизора	Явный запрос	Вызов функции операционной системы

- **Прерывание таймера.** Операционная система определяет, что текущий процесс выполняется в течение максимально разрешенного промежутка времени, именуемого **квантом времени** (time slice). Квант времени представляет собой максимальное количество времени, которое процесс может выполняться без прерывания. Если это так, то данный процесс нужно переключить в состояние готовности и передать управление другому процессу.
- **Прерывание ввода-вывода**. Операционная система определяет, что именно произошло, и если это то событие, которого ожидают один или несколько процессов, операционная система переводит все соответствующие блокированные процессы в состояние готовности (соответственно, блокированные/приостановленные процессы она переводит в состояние готовых/приостановленных процессов). Затем операционная система должна принять решение: возобновить выполнение текущего процесса или передать управление готовому к выполнению процессу с более высоким приоритетом.
- **Ошибка отсутствия блока в памяти**. Допустим, что процессор должен обратиться к слову виртуальной памяти, которое в настоящий момент отсутствует в основной памяти. При этом операционная система должна загрузить в основную память блок (страницу или сегмент), в котором содержится адресованное слово. Сразу же после запроса на загрузку блока операционная система может передать управление другому процессу, а процесс, для продолжения выполнения которого нужно загрузить блок в основную память, переходит в блокированное состояние. После загрузки нужного блока этот процесс переходит в состояние готовности.

28. Процессы в ОС UNIX SVR4. Диаграмма состояний, основные структуры.

Состояния процессов

Всего в операционной системе UNIX SVR4 распознается девять состояний процессов, перечисленных в табл. 3.9; соответствующая диаграмма переходов состояний показана на рис. 3.17 (в ее основе — рисунок из [14]). Этот рисунок похож на рис. 3.9, *б*; нужно только принять во внимание, что два спящих состояния в системе UNIX соответствуют двум блокированным состояниям. Кратко перечислим основные различия между диаграммами.

Таблица 3.9. Состояния процессов в UNIX

Выполняющийся пользовательский	Выполняющийся в пользовательском режиме
Выполняющийся ядра	Выполняющийся в режиме ядра
Готовый к выполнению, загруженный	Готов к выполнению, как только ядро решит передать ему управление
Спящий, загруженный	Не может выполняться, пока не произойдет некоторое событие; процесс находится в основной памяти (блокированное состояние)
Готовый к выполнению, выгруженный	Процесс готов к выполнению, но прежде чем ядро сможет спланировать его запуск, процесс свопинга должен загрузить этот процесс в основную память
Спящий, выгруженный	Процесс ожидает некоторого события; он выгружен из основной памяти (блокированное состояние)
Вытесненный	В момент переключения процессора из режима ядра в пользовательский режим ядро решает передать управление другому процессу
Созданный	Процесс только что создан и еще не готов к выполнению
Зомби	Самого процесса больше не существует, но записи о нем остались с тем, чтобы ими мог воспользоваться родительский процесс

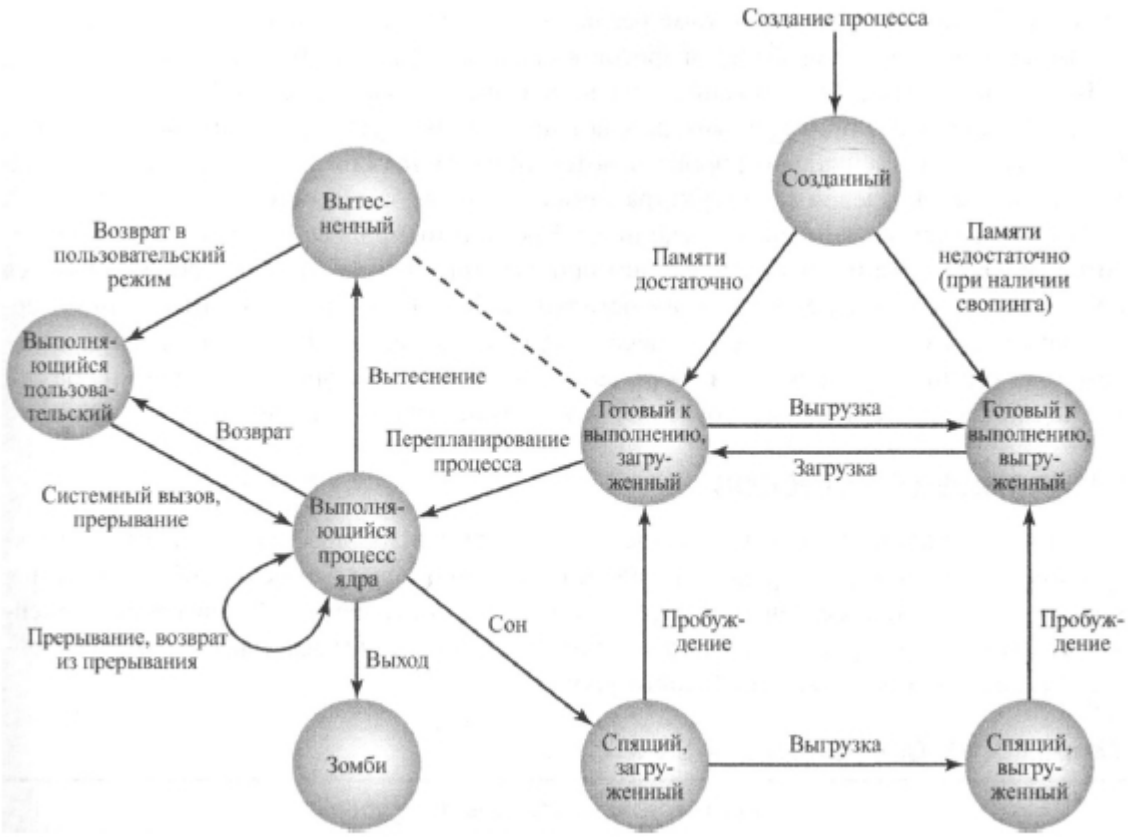
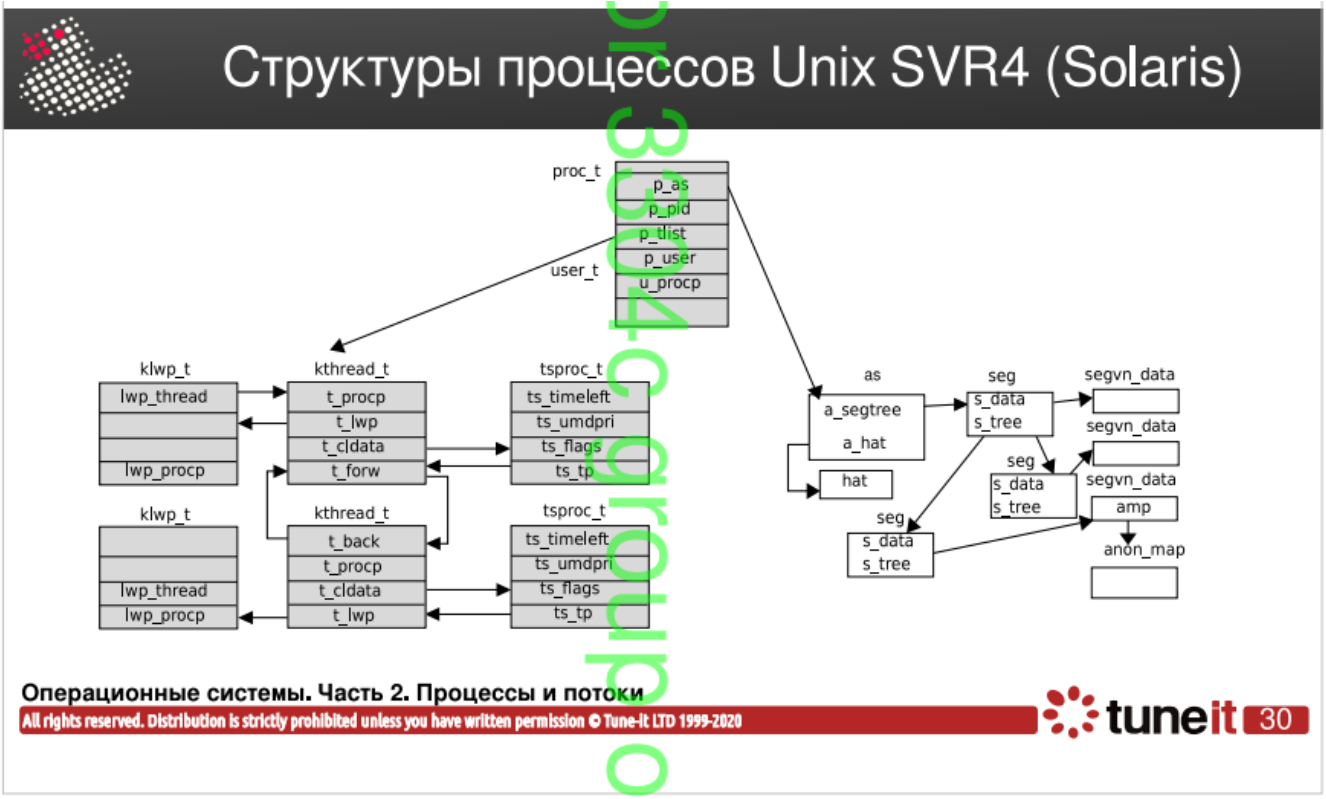


Рис. 3.17. Диаграмма переходов между состояниями процессов в системе UNIX



На слайде представлена небольшая схема, демонстрирующая то, как связаны между собой структуры PCB в Unix SVR4 (Solaris).

В Solaris PCB структурно состоит из двух частей: proc_t и user_t. Эти структуры как бы вставлены друг в друга. proc_t и user_t (т.е. PCB) занимают суммарно порядка 2Кб памяти.

p_as – это указатель на адресацию памяти.

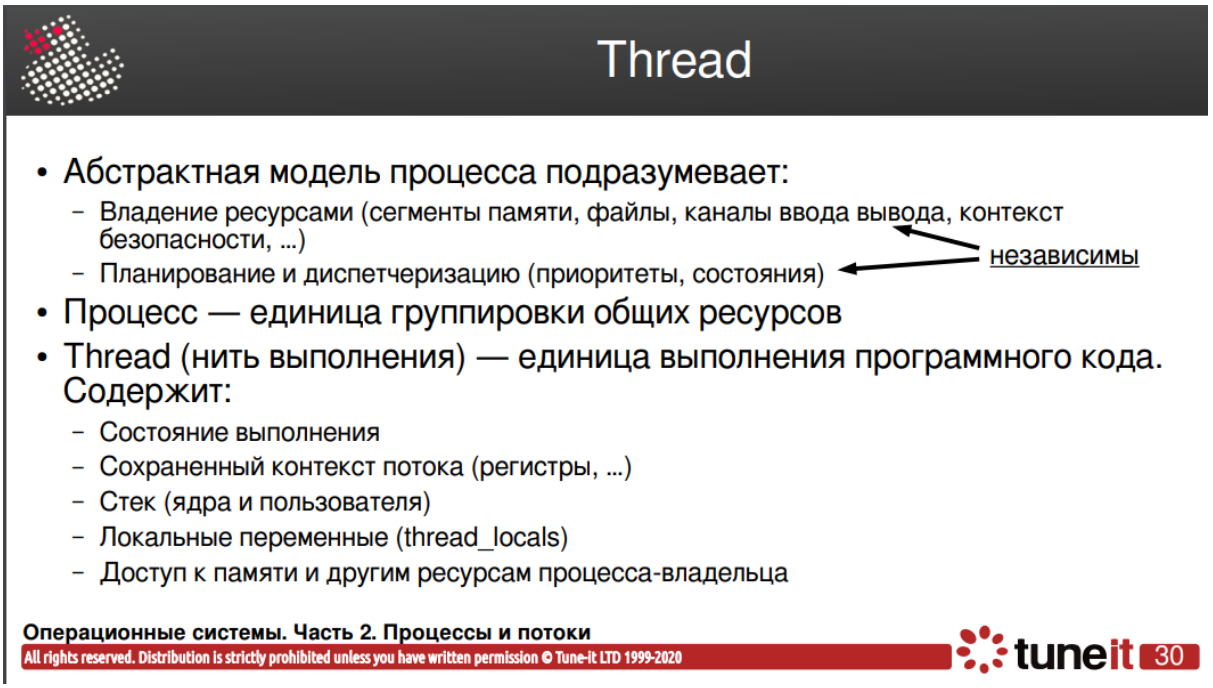
На схеме также показаны структуры, относящиеся к адресному пространству; структуры, относящиеся к конкретному сегменту адресного пространства; а также структуры, которые говорят о том именованное это адресное пространство, или анонимное. Признак именованности/неименованности означает, что адресное пространство связано с файлом или с ним не связано. Например: именованное пространство – это сегмент данных в файле, анонимное пространство – это, например «куча» или стек.

Внутри процесса есть такая абстракция как трэды. На схеме представлены трэды на уровне ядра и трэды на уровне пользователя. Обе эти категории трэдов отдельно диспетчеризуются и у них существуют специальные приоритеты. Все это связано между собой в некоторое количество структур.

На схеме приведены далеко не все существующие поля, так как различных полей, описывающих структуры процесса боле сотни. Причем, чем глубже происходит изучение структуры, тем больше она начинает «разрастаться». Это объясняется тем, что ядро – это достаточно сложный объект для изучения.

Самое «неприятное» здесь в том, что большое количество переменных от версии к версии меняется. И если на уровне пользователя (через POSIX) операционная система представляется единым целым, то в действительности, всё содержание от версии к версии «дышит». Поэтому перед тем, как начинать заниматься отладкой необходимо внимательно загрузить именно необходимые версии всех используемых компонентов.

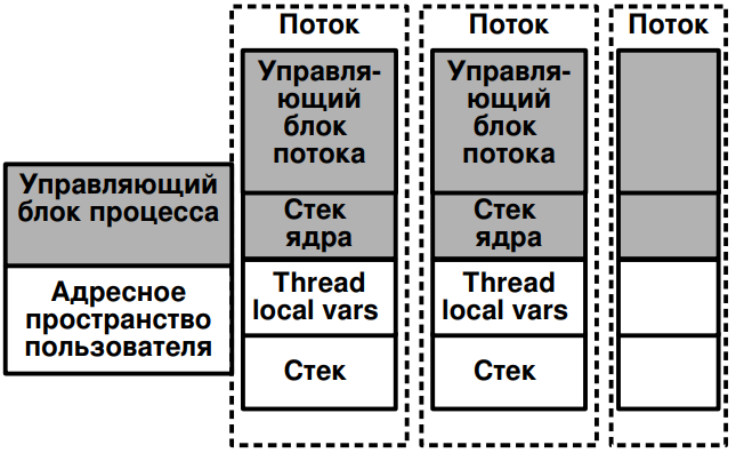
29. Понятие потока выполнения, связь потока и процесса. Преимущества потоков.



Связь структур ядра процесса и потока

Однопоточная модель

Многопоточная модель

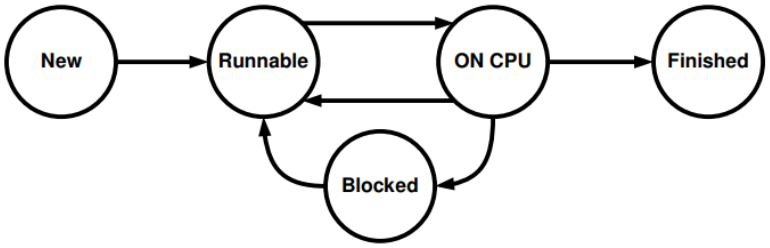


Преимущество потоков

- Общее преимущества в быстродействии:
  - Потоки создаются на порядок быстрее
  - Потоки переключаются быстрее
  - Потоки завершаются быстрее
  - Потоки могут обмениваться информацией быстрее (без участия ядра)
- Даже в однопроцессорной системе есть преимущества
  - Работа в приоритетном и фоновом режиме
  - Асинхронная обработка частей программы
  - Модульная структура программы

30. Состояния потока, User Level Threads vs Kernel Level Threads

Состояние потоков



- NB! потоки используют адресное пространство процесса → необходима синхронизация по общим данным!
- Блокирование потока не должно приводить к блокированию процесса

User Level Threads vs Kernel Level Threads

- ULT (Green Threads) — реализуются библиотеками (или приложениями) на стороне пользователя
- KLT (иногда LWP — light-weight processes) — реализуются ядром

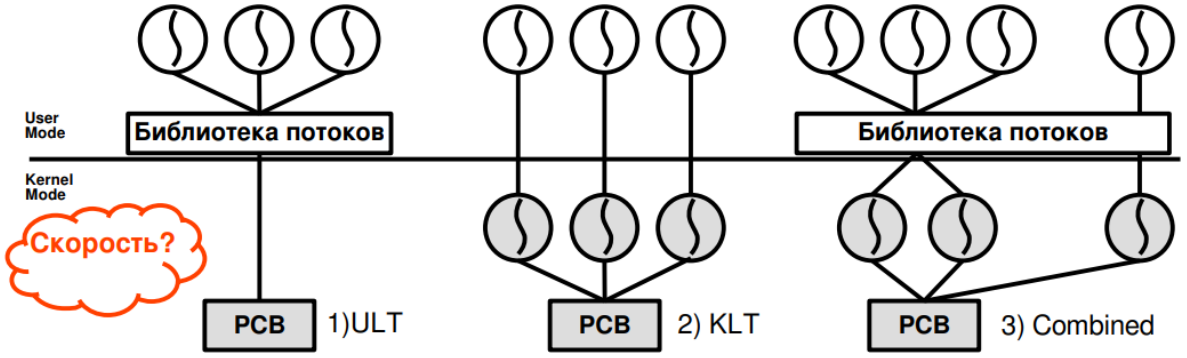


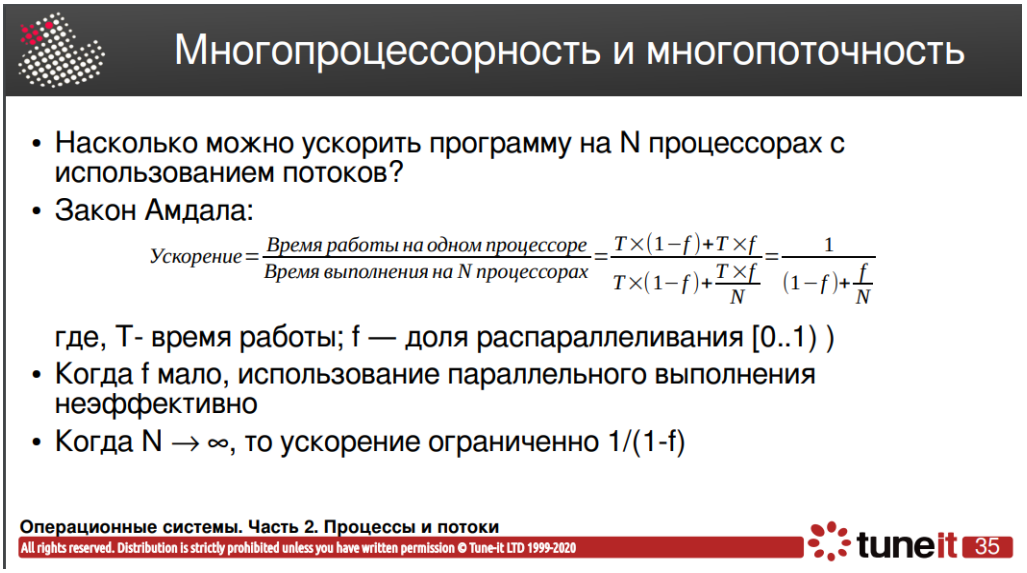
Таблица 4.1. Время задержек потоков (мс)

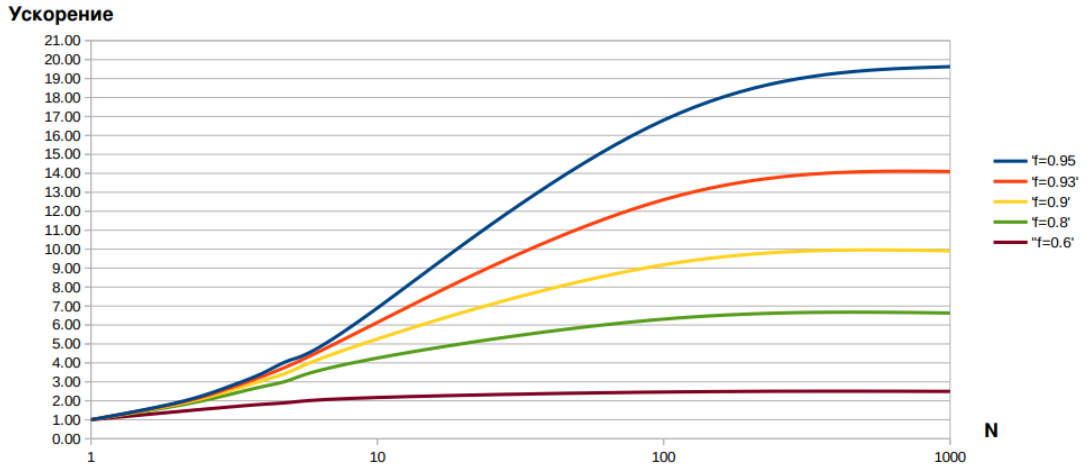
Операция	Пользовательские потоки	Потоки на уровне ядра	Процессы
Нулевое ветвление	34	948	11300
Ожидание сигнала	37	441	1840

Таблица 4.2. Соотношение между потоками и процессами

Потоки: процессы	Описание	Примеры систем
1:1	Каждый поток реализован в виде отдельного процесса с собственным адресным пространством и со своими ресурсами	Традиционные реализации системы UNIX
M:1	Для процесса задаются адресное пространство и динамическое владение ресурсами. В рамках этого процесса может быть создано несколько потоков	OS/2, Windows NT, Solaris, Linux, OS/390, MACH
1:M	Поток может переходить из среды одного процесса в среду другого процесса. Это облегчает перенос потоков из одной системы в другую	Ra (Clouds), Emerald
M:N	Сочетает в себе подходы, основанные на соотношениях M:1 и 1:M	TRIX

31. Многопроцессорность и многопоточность. Закон Амдала.





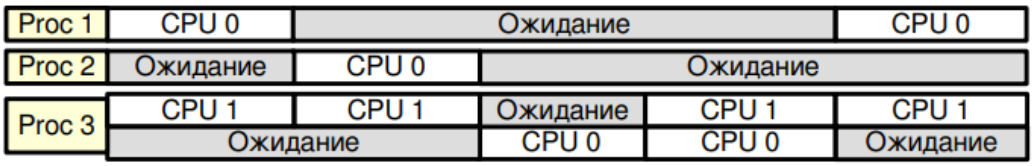
32. Механизм параллельных вычислений, функции ОС.

Механизм «параллельных» вычислений

- Однопроцессорные системы — процессы чередуются

Время

- Многопроцессорные — чередуются и перекрываются.
  - Конкуренция за общие ресурсы. Голодание.



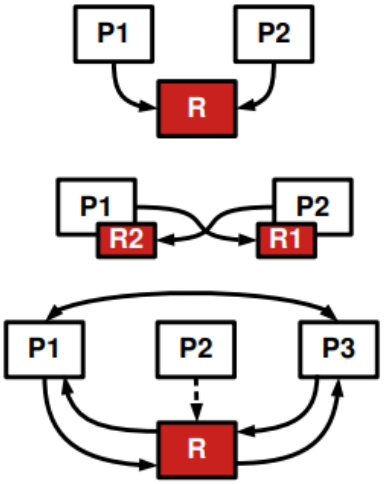
Требуемые функции ОС

- Отслеживание ресурсов процесса/потока
- Распределение и освобождение ресурсов для каждого активного процесса/потока (CPU, файлы, память, ...)
- Защита ресурсов процесса/потока от непреднамеренного воздействия других процессов/потоков
- Независимость результата процесса/потока от скорости его выполнения и скорости других процессов/потоков

33. Проблемы параллельного выполнения: взаимоисключения, взаимоблокировки, голодание. Требования к взаимным исключениям. Уровни взаимодействия процессов и потоков.

Проблемы параллельного выполнения

- Взаимоисключения (Mutual Exclusion) — процессы/потоки не должны одновременно использовать критический ресурс
- Взаимоблокировки (DeadLocks, LiveLocks) — процессы/потоки не должны взаимозахватывать требуемые ресурсы.
- Голодание (Starvation) — конкуренция за ресурсы не должна порождать невозможность доступа к ресурсу



Требования к взаимным исключениям

- Взаимоисключения осуществляются в принудительном порядке.
  - В критическом участке кода должен находится один процесс/поток
- Процесс/поток не должен влиять на другие процессы/потоки в некритическом участке
- Противодействие бесконечному ожиданию доступа к критическому участку
- Вход в свободный критический участок должен незамедлительно предоставляться
- Отсутствие предположений о количестве процессов или их скорости
- Ограничение времени нахождения в критических участках

- **Процессы не осведомлены о наличии друг друга**. Это независимые процессы, не предназначенные для совместной работы. Наилучшим примером такой ситуации может служить многозадачность множества независимых процессов. Это могут быть пакетные задания, интерактивные сессии или комбинация тех и других. Хотя эти процессы и не работают совместно, операционная система должна решать вопросы **конкурентного** использования ресурсов. Например, два независимых приложения могут затребовать доступ к одному и тому же диску или к принтеру. Операционная система должна регулировать такие обращения.
- **Процессы косвенно осведомлены о наличии друг друга**. Эти процессы не обязательно должны быть осведомлены о наличии друг друга с точностью до идентификатора процесса, однако они совместно обращаются к некоторому объекту, например к буферу ввода-вывода. Такие процессы демонстрируют **сотрудничество** при совместном использовании общего объекта.
- **Процессы непосредственно осведомлены о наличии друг друга**. Такие процессы способны общаться один с другим с использованием идентификаторов процессов и изначально созданы для совместной работы. Также они демонстрируют **сотрудничество** при работе.

34. Примитивы синхронизации ОС. Предназначение примитивов синхронизации

Таблица 5.3. Основные механизмы параллельных вычислений

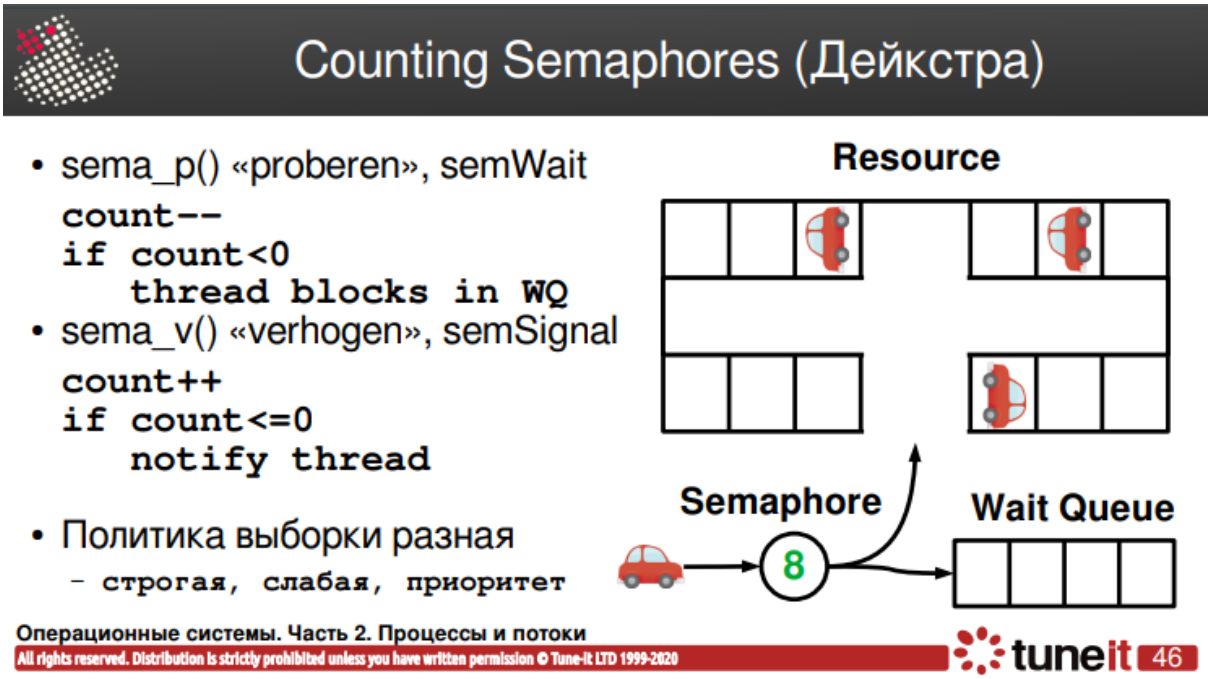
Семафор (Semaphore)	Целочисленное значение, используемое для передачи сигналов между процессами. Над семафором могут быть выполнены только три операции (все они являются атомарными): инициализация, уменьшение (декремент) и увеличение (инкремент) значения. Операция уменьшения может привести к блокировке процесса, а операция увеличения — к разблокированию. Известен также как семафор со счетчиком (counting semaphore) или обобщенный семафор (general semaphore)
Бинарный семафор (Binary semaphore)	Семафор, который может принимать только два значения — 0 и 1
Мьютекс (Mutex)	Аналогичен бинарному семафору. Ключевым отличием является то, что процесс, блокирующий мьютекс (устанавливающий его значение равным 0), должен и разблокировать его (установить его значение равным 1)

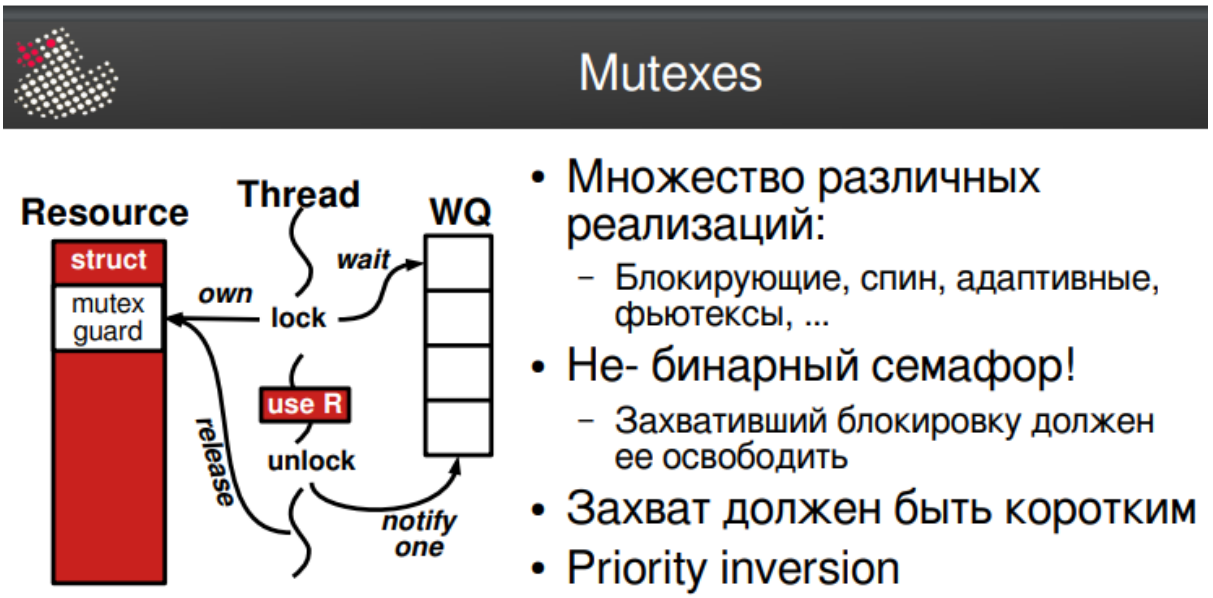
Окончание табл. 5.3

Условная переменная (Condition variable)	Тип данных, используемый для блокировки процесса или потока до тех пор, пока не станет истинным некоторое условие
Монитор (Monitor)	Конструкция языка программирования, инкапсулирующая переменные, процедуры доступа и код инициализации, в абстрактном типе данных. Переменные монитора могут быть доступны только через его процедуры доступа, и в любой момент времени только один процесс может активно работать с монитором. Процедуры доступа представляют собой *критические участки*. Монитор может иметь очередь процессов, ожидающих доступа к нему
Флаги событий (Event flags)	Слово памяти, используемое как механизм синхронизации. Код приложения может связать с каждым битом флага свое событие. Поток может ждать либо одного события, либо сочетания событий путем проверки одного или нескольких битов в соответствующем флаге. Поток блокируется до тех пор, пока все необходимые биты не будут установлены (И) или пока не будет установлен хотя бы один из битов (ИЛИ)
Почтовые ящики/ сообщения (Mailboxes/messages)	Средство обмена информацией между двумя процессами, которое может быть использовано для синхронизации
Спин-блокировки (Spinlocks)	Механизм взаимоисключения, в котором процесс выполняется в бесконечном цикле, ожидая, когда значение блокирующей переменной укажет доступность критического участка

35. Примитивы синхронизации ОС. Семафоры и мьютексы. Бинарный семафор

Семафоры (Semaphore) — захват и освобождение множественного ресурса – или одного (бинарные семафоры)
Мьютексы (Mutex) — блокировка и освобождение ресурса единственным процессом/потоком





36. Примитивы синхронизации ОС. Условные переменные, rwlocks.

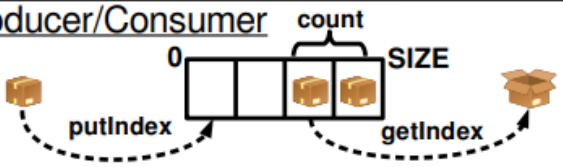
Условные переменные (Conditional Variable) — Блокировка до выполнения какого-либо условия

Блокировки чтения/записи (rw-lock) — отдельные блокировки на чтение и запись

Conditional Variables

- wait(condition, lock)
- signal(condition, lock)
- broadcast(condition, lock)

Producer/Consumer




```
void produce(char c) {
    lock_acquire(&lock);
    while (count == SIZE) {
        cond_wait(&spaceAvailable, &lock);
    }
    count++;
    buffer[putIndex] = c;
    putIndex++;
    if (putIndex == SIZE) {
        putIndex = 0;
    }
    cond_signal(&dataAvailable, &lock);
    lock_release(&lock);
}
```

```
char consume() {
    char c;
    lock_acquire(&lock);
    while (count == 0) {
        cond_wait(&dataAvailable, &lock);
    }
    count--;
    c = buffer[getIndex];
    getIndex++;
    if (getIndex == SIZE) getIndex = 0;
    cond_signal(&spaceAvailable, &lock);
    lock_release(&lock);
    return c;
}
```

Операционные системы. Часть 2. Процессы и потоки

tuneit 48

Multiple-reader, single-writer locks (rwlocks)

- Когда читатели читают, они захватывает readlock, количество одновременных читателей содержится в rwlock
- Когда писателю требуется записать — он устанавливает требование записи (want write) и ожидает на rwlock
- rwlock ждет освобождения readlock
- Оповещает писателей
  - Один захватывает writelock (читатели не могут читать)
- Оповещает читателей
  - Захватывается readlock (писатели должны требовать)

Readers/Writers

Операционные системы. Часть 2. Процессы и потоки

tuneit 49

37. Примитивы синхронизации ОС. Мониторы, флаги событий, передача сообщений.

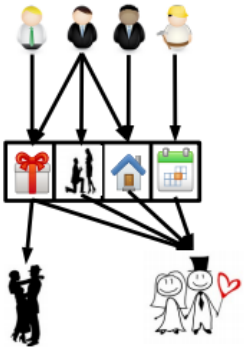
Мониторы (Monitor) — конструкции языков программирования, которые скрывают низкоуровневые примитивы синхронизации
Флаги событий (Event Flags) — связывание условий продолжения выполнения с одним или несколькими флагами (битами блокирующей переменной)
Почтовые ящики (Message Passing) — передача сообщений

Monitors

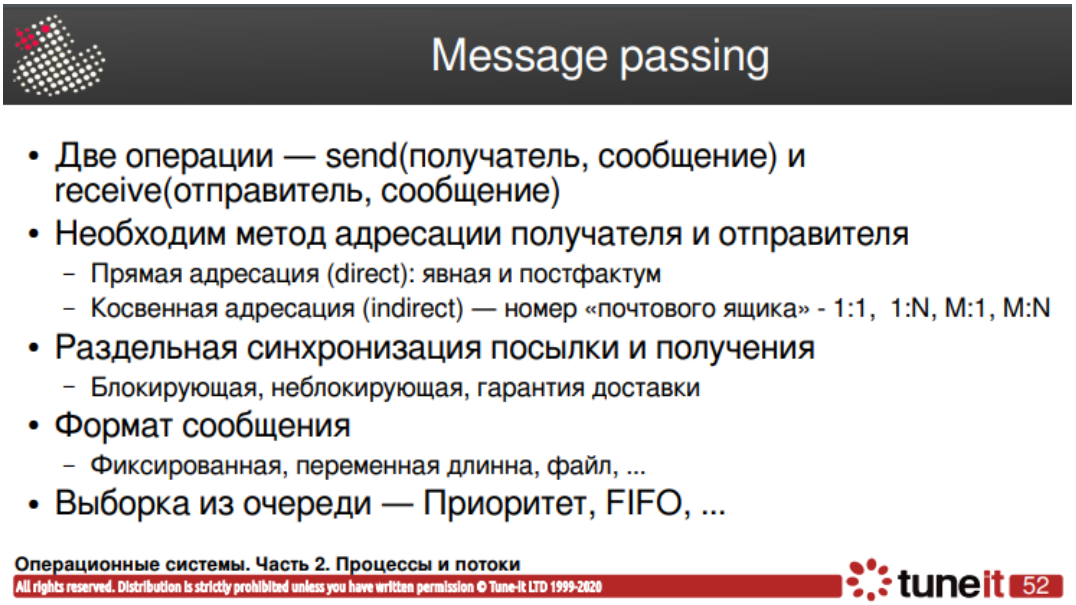
- Набор процедур, выполняющих операции над общими данными
- Каждая процедура подразумевает захват блокировки общих данных
- Локальные переменные используются только внутри монитора
- Для ожидания и оповещения используются Conditional Variables
- Parallel Pascal, Java
- Реализуются высокоуровнево, программисту не требуется возится с захватом/освобождением или ожиданием/нотификацией
- ИМХО В операционных системах не используются

Event Flags

- Наборы (битовых) флагов ожидания событий
- Операции
  - Установка флага
  - Сброс флага
  - Ожидание флага
  - Ожидание какого-либо флага
  - Ожидание всех флагов
- Реализованы VMS, python



связывание условий продолжения выполнения с одним или несколькими флагами (битами блокирующей переменной)



38. Примитивы синхронизации ОС. Неблокирующие примитивы синхронизации и неблокирующие структуры данных.

Блокировка вызывает переключение контекста – Дорого (долго)!

Преимущество неблокирующих алгоритмов — в лучшей масштабируемости по количеству процессоров. К тому же, **если ОС прервёт один из потоков фоновой задачей, остальные, как минимум, выполнят свою работу, не простаивая. По максимуму — возьмут невыполненную работу на себя.**

Сейчас модно пользоваться неблокирующими примитивами синхронизации и неблокирующими структурам – Wait-free (N-steps), lock-free (some N-steps, other retry), obstruction-free

Без препятствий (англ. obstruction-free)
Самая слабая из гарантий. Поток совершает прогресс, если не встречает препятствий со стороны других потоков. Алгоритм работает без препятствий, если поток, запущенный в любой момент (при условии, что выполнение всех препятствующих потоков приостановлено) завершит свою работу за детерминированное количество шагов. Синхронизация с помощью мьютексов не отвечает даже этому требованию: если поток остановится, захватив мьютекс, то остальные потоки, которым этот мьютекс нужен, будут простаивать.

Без блокировок (англ. *lock-free*)

Для алгоритмов без блокировок гарантируется системный прогресс по крайней мере одного потока. Например, поток, выполняющий операцию «сравнение с обменом» в цикле, теоретически может выполняться бесконечно, но каждая его итерация означает, что какой-то другой поток совершил прогресс, то есть система в целом совершает прогресс.

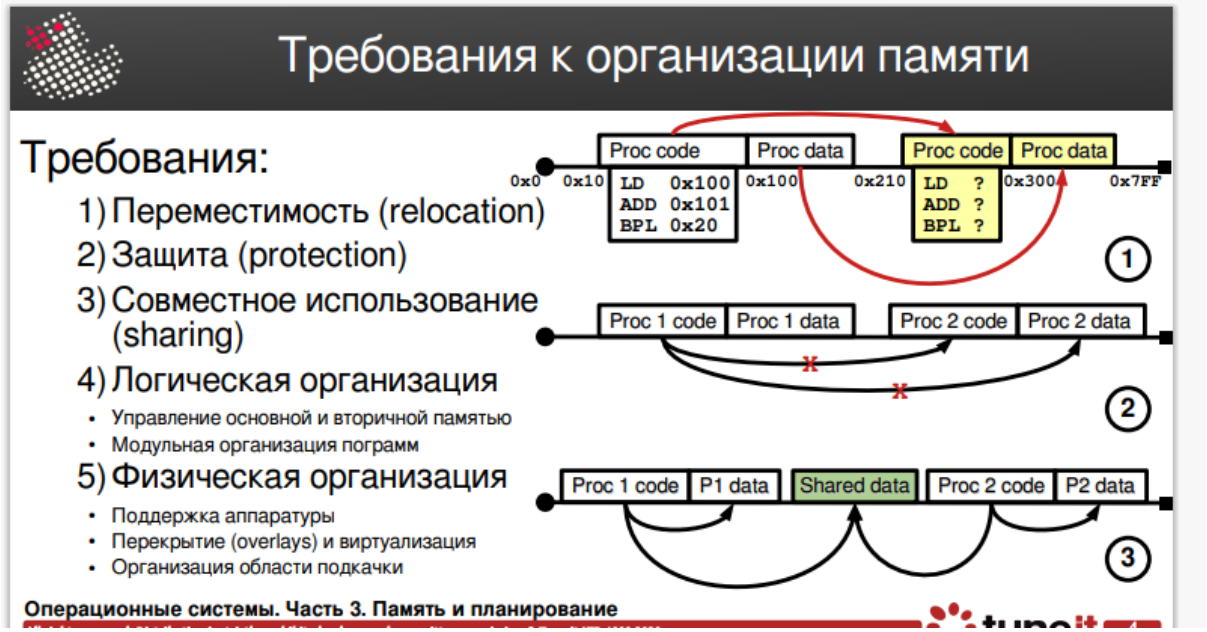
Без ожиданий (англ. *wait-free*)

Самая строгая гарантия прогресса. Алгоритм работает без ожиданий, если каждая операция выполняется за определённое количество шагов, не зависящее от других потоков

Примеры:

- Атомарные операции и примитивы синхронизации
- Неблокирующие структуры данных — списки, деревья, очереди, … over 9000.

39. Управление памятью, основные определения и требования к организации.



Перемещение

В многозадачной системе доступная основная память в общем случае разделяется среди множества процессов. Обычно программист не знает заранее, какие программы будут резидентно находиться в основной памяти во время работы разрабатываемой им программы. Кроме того, для максимизации загрузки процессора желательно иметь большой пул процессов, готовых к выполнению, для чего требуется возможность загрузки и выгрузки активных процессов из основной памяти. Требование, чтобы выгруженная из памяти программа была вновь загружена в то же самое место, где находилась и ранее, было бы слишком сильным ограничением. Крайне желательно,

чтобы программа могла быть перемещена (relocate) в другую область памяти. Таким образом, заранее неизвестно, где именно будет размещена программа, а кроме того, программа может быть перемещена из одной области памяти в другую при свопинге. Эти обстоятельства обусловливают наличие определенных технических требований к адресации, проиллюстрированных на рис. 7.1.

Рис. 7.1. Требования к адресации процесса

На рисунке представлен образ процесса. Для простоты предположим, что образ процесса занимает одну непрерывную область основной памяти. Очевидно, что операционной системе необходимо знать местоположение управляющей информации процесса и стека выполнения, а также точки входа для начала выполнения процесса. Поскольку управлением памятью занимается операционная система и она же размещает процесс в основной памяти, соответствующие адреса она получает автоматически. Однако, помимо получения операционной системой указанной информации, процесс должен иметь возможность обращаться к памяти в самой программе. Так, команды ветвления содержат адреса, указывающие на команды, которые должны быть выполнены после них; команды обращения к данным - адреса байтов или слов, с которыми они работают. Так или иначе, но процессор и программное обеспечение операционной системы должны быть способны перевести ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти.

Защита

Каждый процесс должен быть защищен от нежелательного воздействия других процессов, случайного или преднамеренного. Следовательно, код других процессов не

должен иметь возможности без разрешения обращаться к памяти данного процесса для чтения или записи. Однако удовлетворение требованию перемещаемости усложняет задачу защиты.
Поскольку расположение программы в основной памяти непредсказуемо, проверка абсолютных адресов во время компиляции невозможна. Кроме того, в большинстве языков программирования возможно динамическое вычисление адресов во время выполнения (например, вычисление адреса элемента массива или указателя на поле структуры данных). Следовательно, во время работы программы необходимо выполнять проверку всех обращений к памяти, генерируемых процессом, чтобы удостовериться, что все они - только к памяти, выделенной данному процессу. К счастью, как вы увидите позже, механизмы поддержки перемещений обеспечивают и поддержку защиты. Обычно пользовательский процесс не может получить доступ ни к какой части операционной системы - ни к коду, ни к данным. Код одного процесса не может выполнить команду ветвления, целевой код которой находится в другом процессе. Если не приняты специальные меры, код одного процесса не может получить доступ к данным другого процесса. Процессор должен быть способен прервать выполнение таких команд. Заметим, что требования защиты памяти должны быть удовлетворены на уровне процессора (аппаратного обеспечения), а не на уровне операционной системы (программного обеспечения), поскольку операционная система не в состоянии предвидеть все обращения к памяти, которые будут выполнены программой. Даже если бы такое было возможно, сканирование каждой программы в поиске предлагаемых нарушений защиты было бы слишком расточительным с точки зрения использования процессорного времени. Следовательно, соответствующие возможности аппаратного обеспечения - единственное средство определения допустимости обращения к памяти (данным или коду) во время работы программы

Совместное использование

Любой механизм защиты должен иметь достаточную гибкость для того, чтобы обеспечить возможность нескольким процессам обращаться к одной и той же области основной памяти. Например, если несколько процессов выполняют один и тот же машинный код, то будет выгодно позволить каждому процессу работать с одной и той же копией этого кода, а не создавать собственную. Процессам, сотрудничающим в работе над некоторой задачей, может потребоваться совместный доступ к одним и тем же структурам данных. Система управления памятью должна, таким образом, обеспечивать управляемый доступ к разделяемым областям памяти, при этом никоим образом не ослабляя защиту памяти. Как мы увидим позже, механизмы поддержки перемещений обеспечивают и поддержку совместного использования памяти.

Логическая организация

Практически всегда основная память в компьютерной системе организована как линейное (одномерное) адресное пространство, состоящее из последовательности байтов или слов. Аналогично организована и вторичная память на своем физическом уровне. Хотя такая организация и отражает особенности используемого аппаратного обеспечения, она не соответствует способу, которым обычно создаются программы. Большинство программ организованы в виде модулей, одни из которых неизменны (только для чтения, только для выполнения), а другие содержат данные, которые могут быть изменены. Если операционная система и аппаратное обеспечение компьютера

могут эффективно работать с пользовательскими программами и данными, представленными модулями, то это обеспечивает ряд преимуществ.
1. Модули могут быть созданы и скомпилированы независимо один от другого, при этом все ссылки из одного модуля во второй разрешаются системой во время работы программы.
2. Разные модули могут получить разные степени защиты (только для чтения, только для выполнения) за счет весьма умеренных накладных расходов.
3. Возможно применение механизма, обеспечивающего совместное использование модулей разными процессами.
Основное достоинство обеспечения совместного использования на уровне модулей заключается в том, что они соответствуют взгляду программиста на задачу и, следовательно, ему проще определить, требуется ли совместное использование того или иного модуля. Инструментом, наилучшим образом удовлетворяющим данным требованиям, является сегментация, которая будет рассмотрена в данной главе среди прочих методов управления памятью.

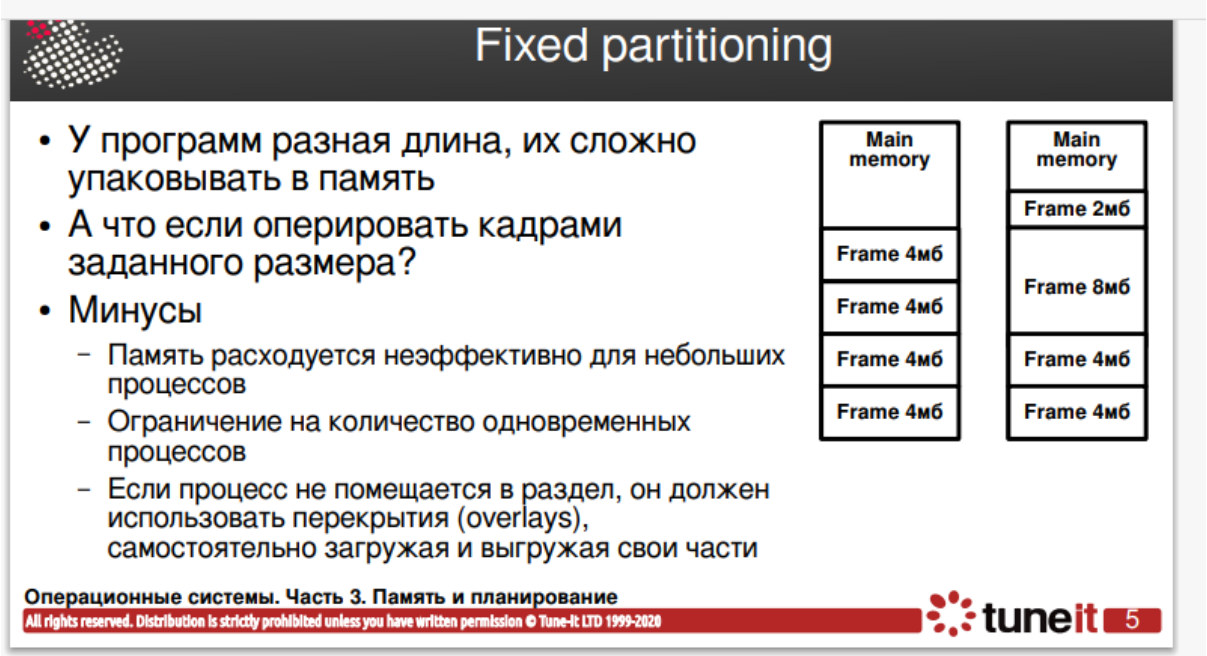
Физическая организация
Как указывалось в разделе 1.5, память компьютера разделяется как минимум на два уровня: основная и вторичная. Основная память обеспечивает быстрый доступ по относительно высокой цене; кроме того, она энергозависима, т.е. не обеспечивает долговременное хранение. Вторичная память медленнее и дешевле основной и обычно не энергонезависима. Следовательно, вторичная память большой емкости может служить для долговременного хранения программ и данных, а основная память меньшей емкости - для хранения программ и данных, использующихся в текущий момент. В такой двухуровневой структуре основной заботой системы становится организация потоков информации между основной и вторичной памятью. Ответственность за эти потоки может быть возложена и на отдельного программиста, но это непрактично и нежелательно по следующим причинам.
1. Основной памяти может быть недостаточно для программы и ее данных. В этом случае программист вынужден прибегнуть к практике, известной как структуры с перекрытием - оверлеи (overlay), когда программа и данные организованы таким образом, что различные модули могут быть назначены одной и той же области памяти; основная программа при этом ответственна за перезагрузку модулей при необходимости. Даже при помощи соответствующего инструментария компиляции оверлеев разработка таких программ приводит к дополнительным затратам времени программиста.
2. Во многозадачной среде программист при разработке программы не знает, какой объем памяти будет доступен программе и где эта память будет располагаться.
Таким образом, очевидно, что задача перемещения информации между двумя уровнями памяти должна возлагаться на операционную систему. Эта задача является сущностью управления памятью.

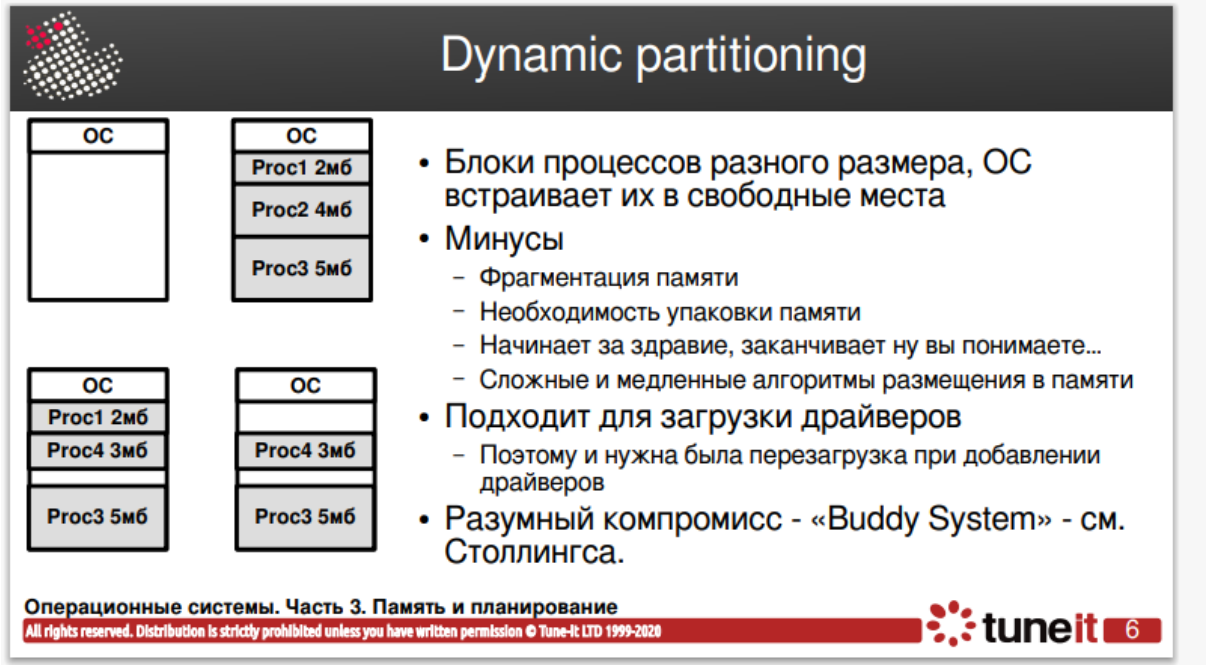
кратко по распределению памяти

Таблица 7.2. Технологии управления памятью

Технология	Описание	Сильные стороны	Слабые стороны
Фиксированное распределение	Основная память разделяется на ряд статических разделов во время генерации системы. Процесс может быть загружен в раздел равного или большего размера	Простота реализации, малые системные накладные расходы	Неэффективное использование памяти из-за внутренней фрагментации, фиксированное максимальное количество активных процессов
Динамическое распределение	Разделы создаются динамически; каждый процесс загружается в раздел строго необходимого размера	Отсутствует внутренняя фрагментация, более эффективное использование основной памяти	Неэффективное использование процессора из-за необходимости уплотнения для противодействия внешней фрагментации
Простая страничная организация	Основная память разделена на ряд кадров равного размера. Каждый процесс разделен на некоторое количество страниц равного размера и той же длины, что и кадры. Процесс загружается путем загрузки всех его страниц в доступные, но не обязательно последовательные кадры	Отсутствует внешняя фрагментация	Наличие небольшой внутренней фрагментации
Простая сегментация	Каждый процесс распределен на ряд сегментов. Процесс загружается путем загрузки всех своих сегментов в динамические (не обязательно смежные) разделы	Отсутствует внутренняя фрагментация; по сравнению с динамическим распределением повышенная эффективность использования памяти и сниженные накладные расходы	Внешняя фрагментация
Страничная организация виртуальной памяти	Все, как при простой страничной организации, с тем исключением, что не требуется одновременно загружать все страницы процесса. Необходимые нерезидентные страницы автоматически загружаются в память	Нет внешней фрагментации; более высокая степень многозадачности; большое виртуальное адресное пространство	Накладные расходы из-за сложности системы управления памятью
Сегментация виртуальной памяти	Все, как при простой сегментации, с тем исключением, что не требуется одновременно загружать все сегменты процесса. Необходимые нерезидентные сегменты автоматически загружаются в память	Нет внутренней фрагментации; более высокая степень многозадачности; большое виртуальное адресное пространство; поддержка защиты и совместного использования	Накладные расходы из-за сложности системы управления памятью

40. Фиксированное и динамическое размещение программ в памяти.





подробнее см столлингса страницы 405-411.

Система двойников

Как фиксированное, так и динамическое распределение памяти имеют свои недостатки. Фиксированное распределение ограничивает количество активных процессов и неэффективно использует память при несоответствии между размерами разделов и процессов. Динамическое распределение реализуется более сложно и включает накладные расходы по уплотнению памяти. Интересным компромиссом в этом плане является система двойников ([136], [191]).

В системе двойников память распределяется блоками размером 2^K, $L \le K \le U$, где

2^L — минимальный размер выделяемого блока памяти;

2^U — наибольший распределяемый блок; вообще говоря, 2^U представляет собой размер всей доступной для распределения памяти.

Вначале все доступное для распределения пространство рассматривается как единый блок размером 2^U. При запросе размером s, таким, что $2^{U-1} < s \le 2^U$, выделяется весь блок. В противном случае блок разделяется на два эквивалентных двойника с размерами 2^{U-1}. Если $2^{U-2} < s \le 2^{U-1}$, то по запросу выделяется один из двух двойников; в противном случае один из двойников вновь делится пополам. Этот процесс продолжается до тех пор, пока не будет сгенерирован наименьший блок, размер которого не меньше s. Система двойников постоянно ведет список "дыр" (доступных блоков) для каждого размера 2^i. Дыра может быть удалена из списка (i+1) разделением ее пополам и внесением двух новых дыр размером 2^i в список i. Когда пара двойников в списке i оказывается освобожденной, они удаляются из списка и объединяются в единый блок в списке (i+1). Ниже приведен рекурсивный алгоритм для удовлетворения запроса размером $2^{i-1} < k \le 2^i$, в котором осуществляется поиск дыры размером 2^i.

```
void get_hole(int i)
{
    if (i == (U+1))  < Ошибка >;
    if (< Список i пуст >)
    {
        get_hole(i+1);
        < Разделить дыру на двойники >;
        < Поместить двойники в список i >;
    }
    < Взять первую дыру из списка i >;
}
```

На рис. 7.6 приведен пример использования блока с начальным размером 1 Мбайт. Первый запрос A — на 100 Кбайт (для него требуется блок размером 128 Кбайт). Для этого начальный блок делится на два двойника по 512 Кбайт. Первый из них делится на двойники размером 256 Кбайт, и, в свою очередь, первый из получившихся при этом разделении двойников также делится пополам. Один из получившихся двойников размером 128 Кбайт выделяется запросу A. Следующий запрос B требует 256 Кбайт. Такой блок имеется в наличии и выделяется. Процесс продолжается с разделением и слиянием двойников при необходимости. Обратите внимание, что после освобождения блока E происходит слияние двойников по 128 Кбайт в один блок размером 256 Кбайт, который, в свою очередь, тут же сливается со своим двойником.

На рис. 7.7 показано представление системы двойников в виде бинарного дерева, непосредственно после освобождения блока B. Листья представляют текущее распределение памяти. Если два двойника являются листьями, то по крайней мере один из них занят; в противном случае они должны слиться в блок большего размера.

Система двойников представляет собой разумный компромисс для преодоления недостатков схем фиксированного и динамического распределения, но в современных операционных системах ее превосходит виртуальная память, основанная на страничной организации и сегментации. Однако система двойников нашла применение в параллельных системах как эффективное средство распределения и освобождения параллельных программ (см., например, [118]). Модифицированная версия системы двойников используется для распределения памяти ядром UNIX (подробнее об этом вы узнаете в главе 8, "Виртуальная память").

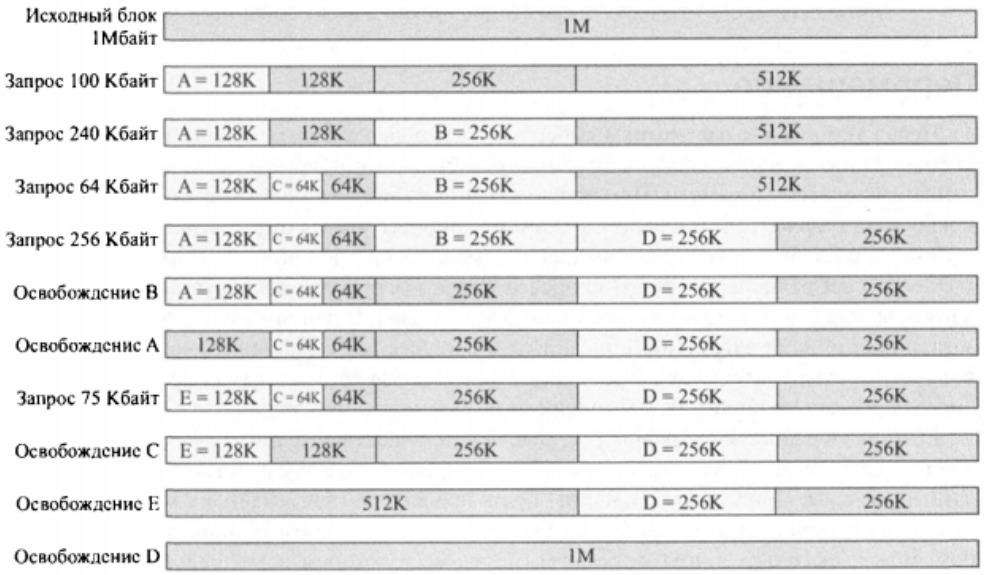


Рис. 7.6. Пример системы двойников

41. Модели аппаратного перемещения программ.

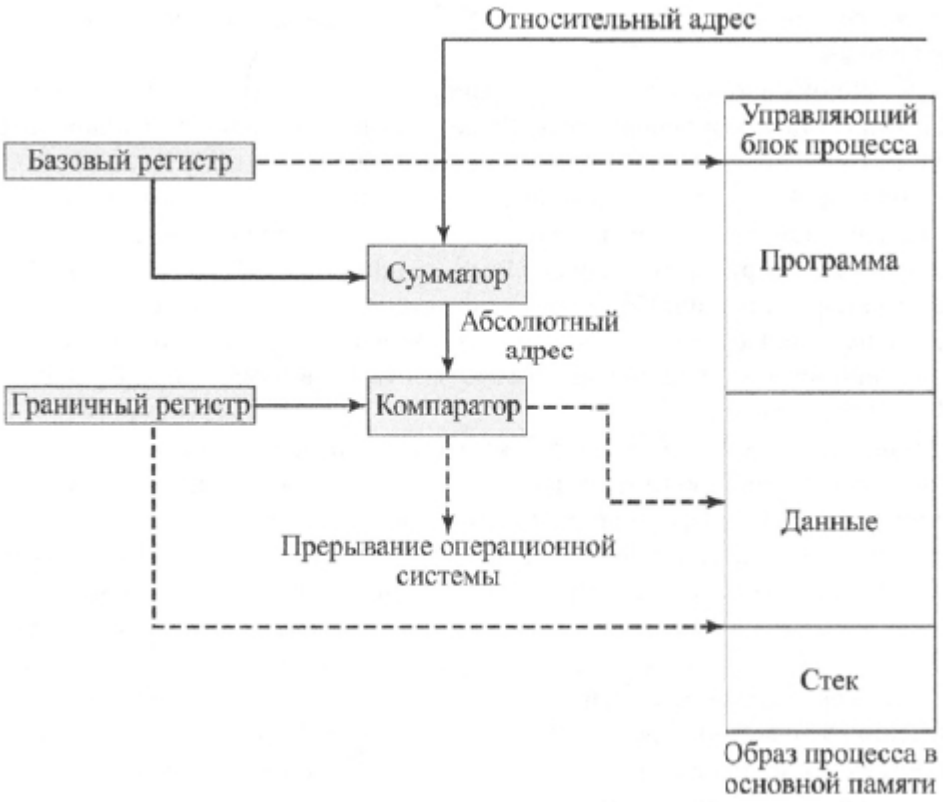
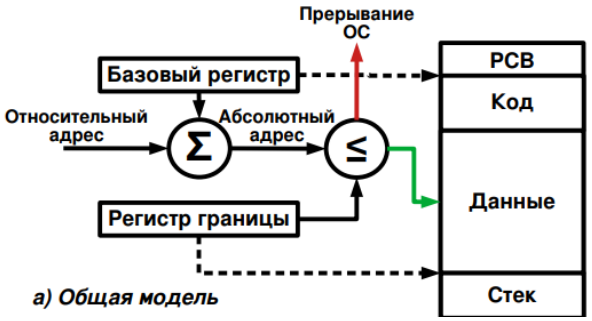
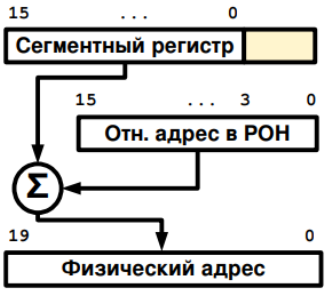


Рис. 7.8. Аппаратная поддержка перемещения

- Копировать программы средствами ОС — долго!
  - Нужна аппаратная поддержка, например, как в 8086
- Модели аппаратного перемещения:

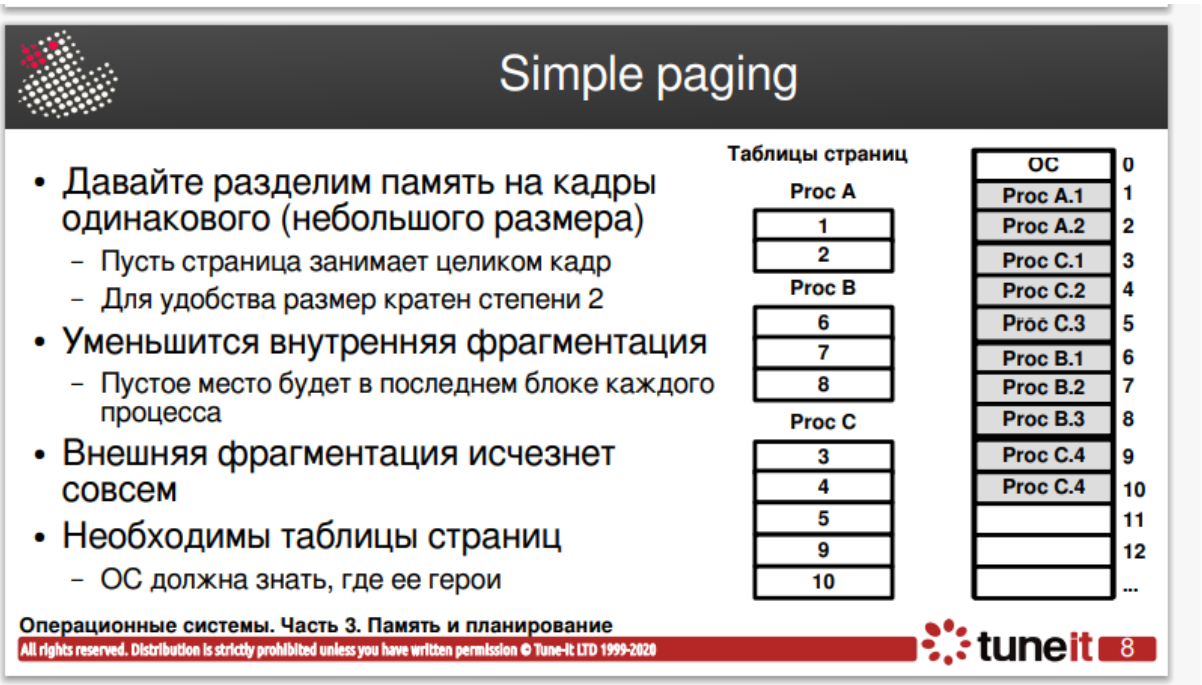


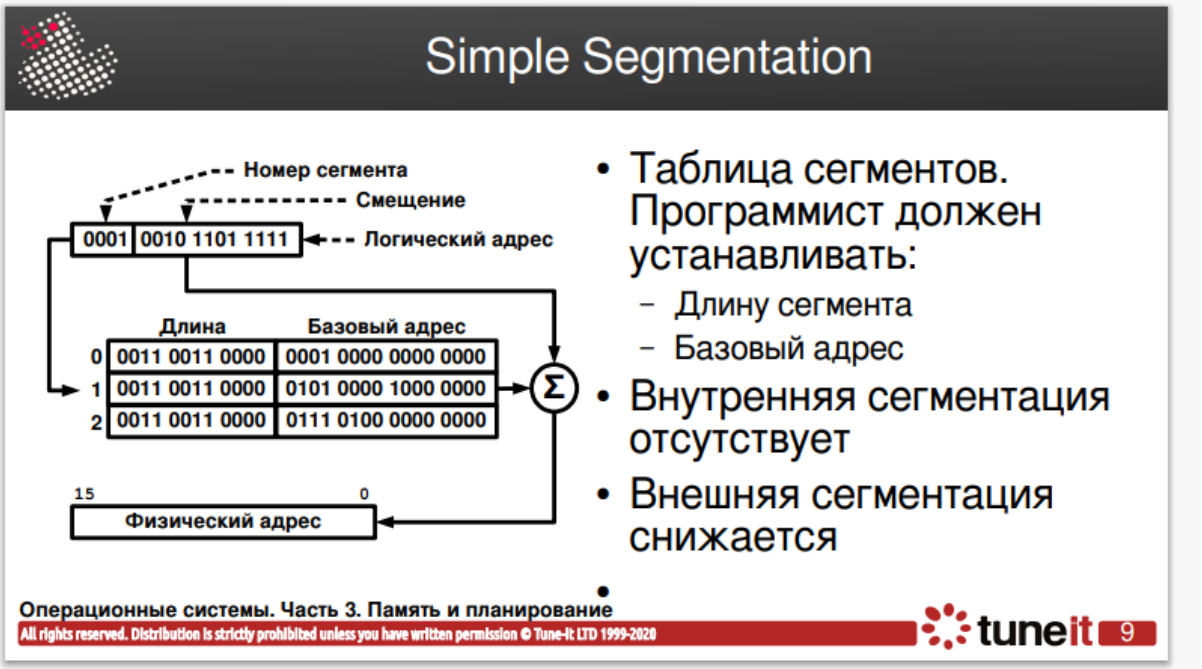
а) Общая модель



б) Модель 8086

42. Простой страничный поход и простая сегментная организация.





43. Виртуальная память основные определения и принципы организации аппаратуры и управляющих программ.

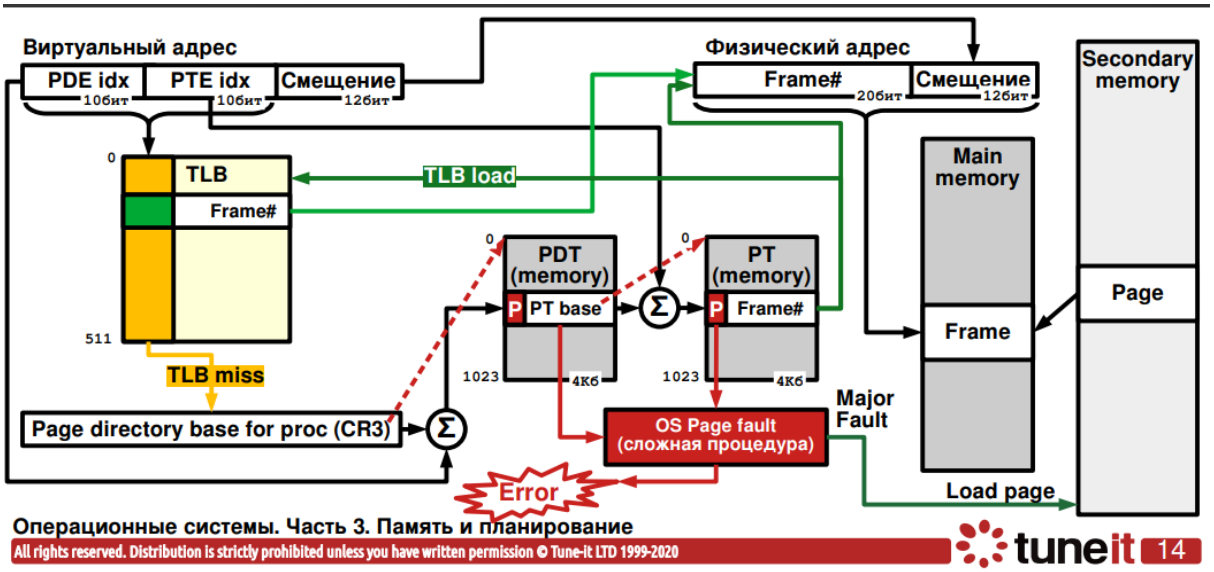
Реальный (физический) адрес — адрес в основной памяти
Виртуальный адрес — логический адрес внутри процесса
Адресное пространство — диапазон адресов процесса

Виртуальное адресное пространство — область для одного процесса с виртуальными адресами
Виртуальная память — схема расположения процессов в памяти, в которой:
– Вторичная память адресуется так-же как и основная
– Виртуальные адреса транслируются в адреса в основной памяти
– Основная память может быть расширена на вторичную
– Размер памяти ограничен схемой адресации, но не фактическим количеством ячеек
Организация виртуальной памяти это совокупность аппаратных и программных средств
Схемы реализации
— совокупность страничной и сегментной адресации

- – Логические адреса динамически транслируются в физические

– Сегменты и страницы не должны располагаться последовательно в основной памяти
– Любая часть процесса в разные моменты выполнения может находится во вторичной памяти и менять физические адреса в основной
Resident set (резидентная часть) — часть процесса, находящаяся в основной памяти
Real memory — часть памяти, где процесс непосредственно выполняется
Virtual memory — часть памяти, которую процесс может занять
Следовательно, ОС может одновременно с бОльшим количеством процессов

44. Виртуальный страничный обмен. Двухуровневая организация MMU и TLB 80386. (для КОТ и ГТ — общие принципы)



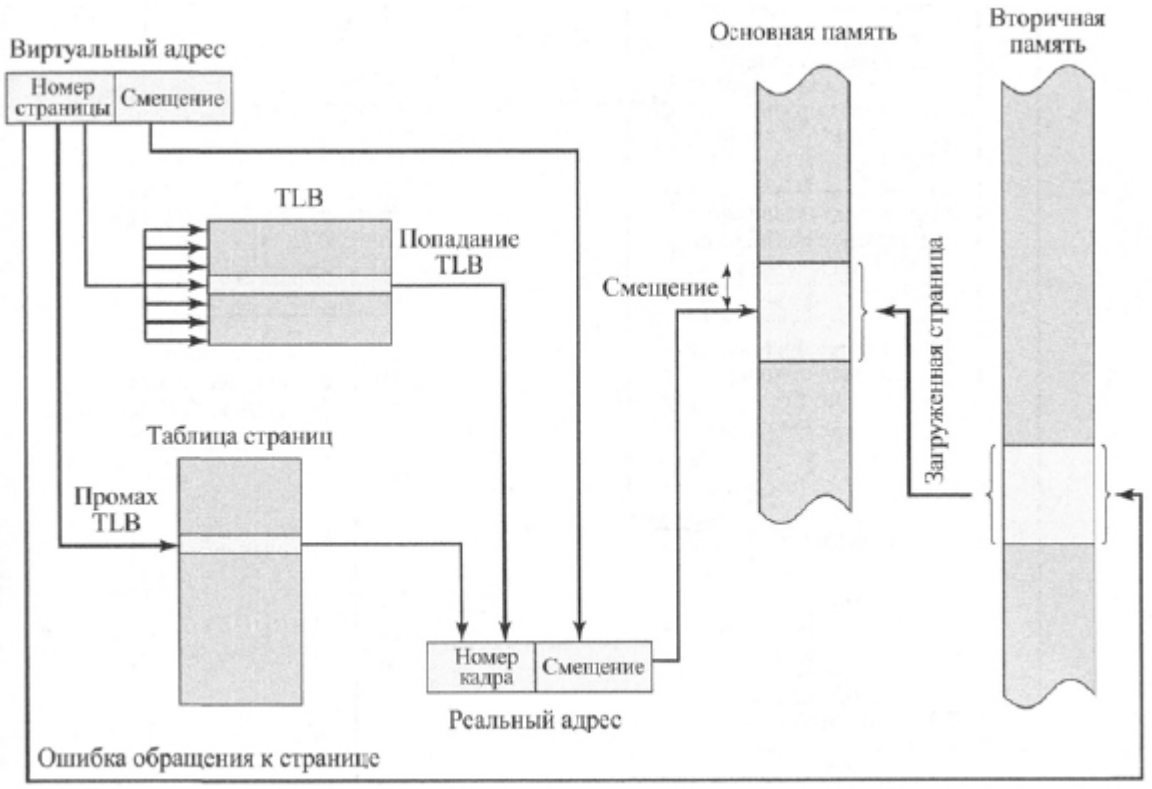
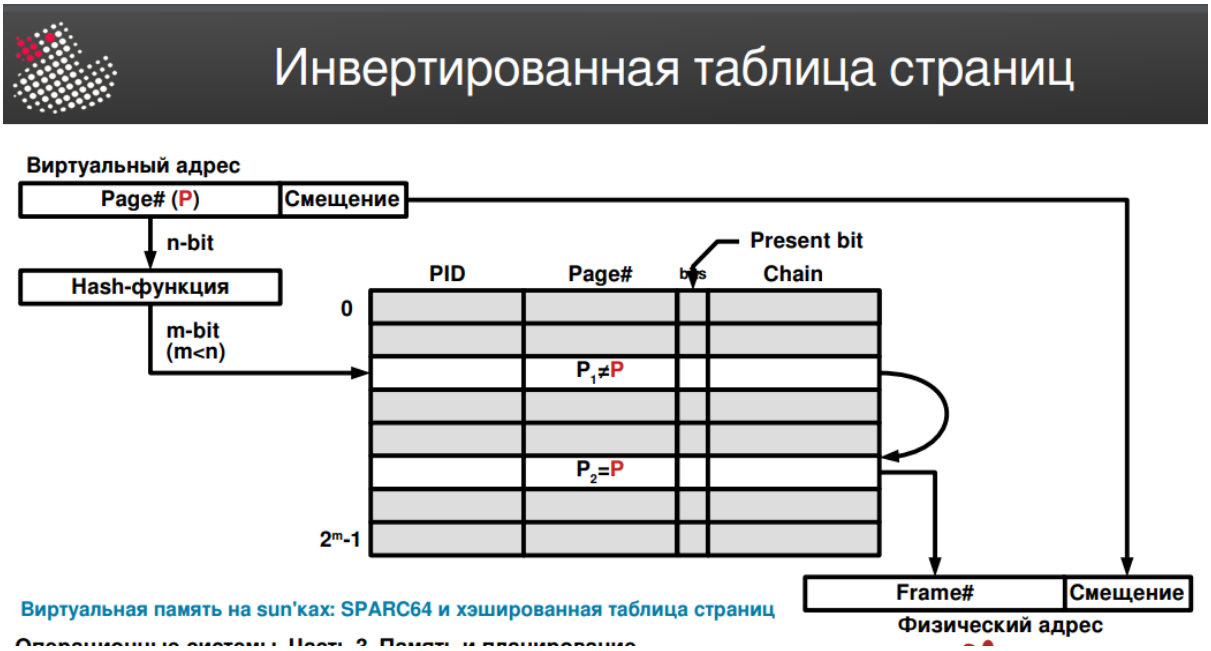
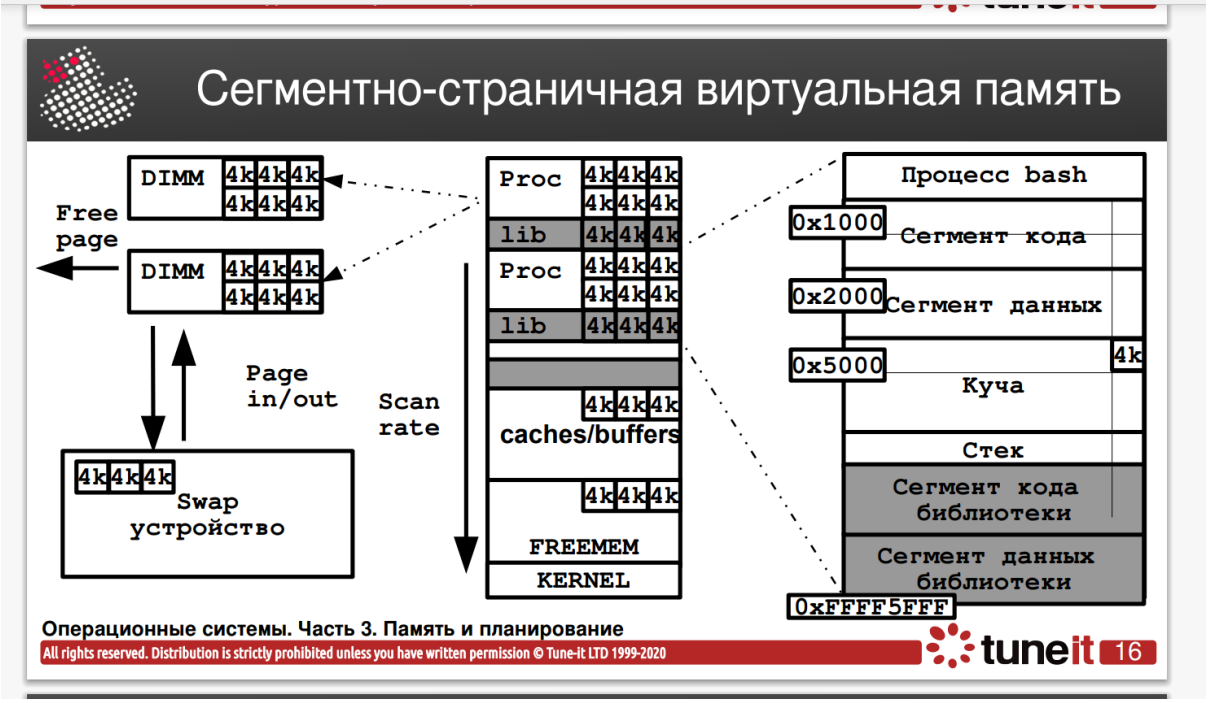


Рис. 8.6. Использование TLB

45. Инвертированная таблица страниц.



46. Сегментно-страничная виртуальная память.



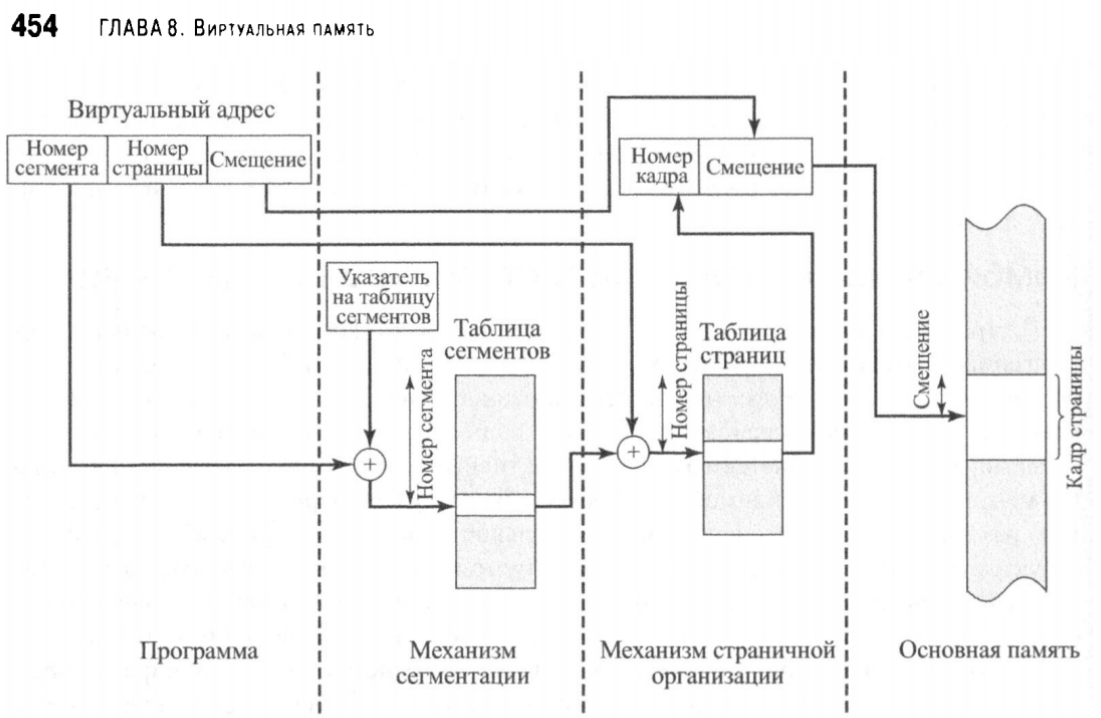
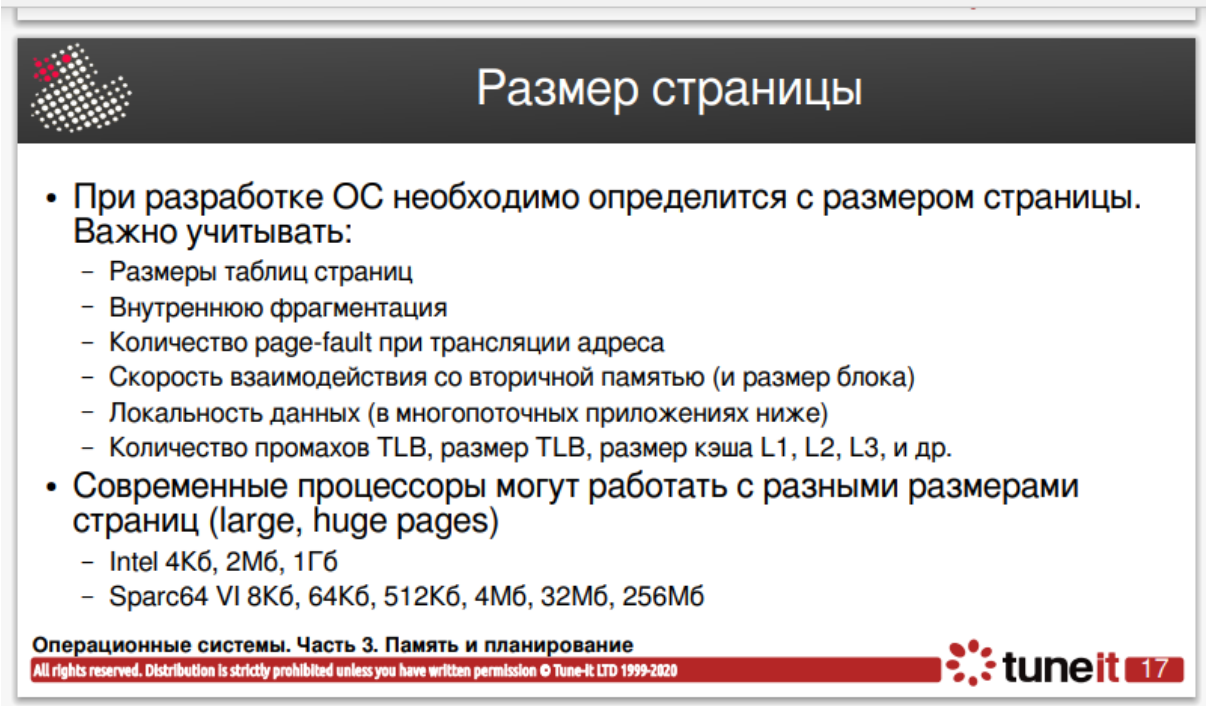
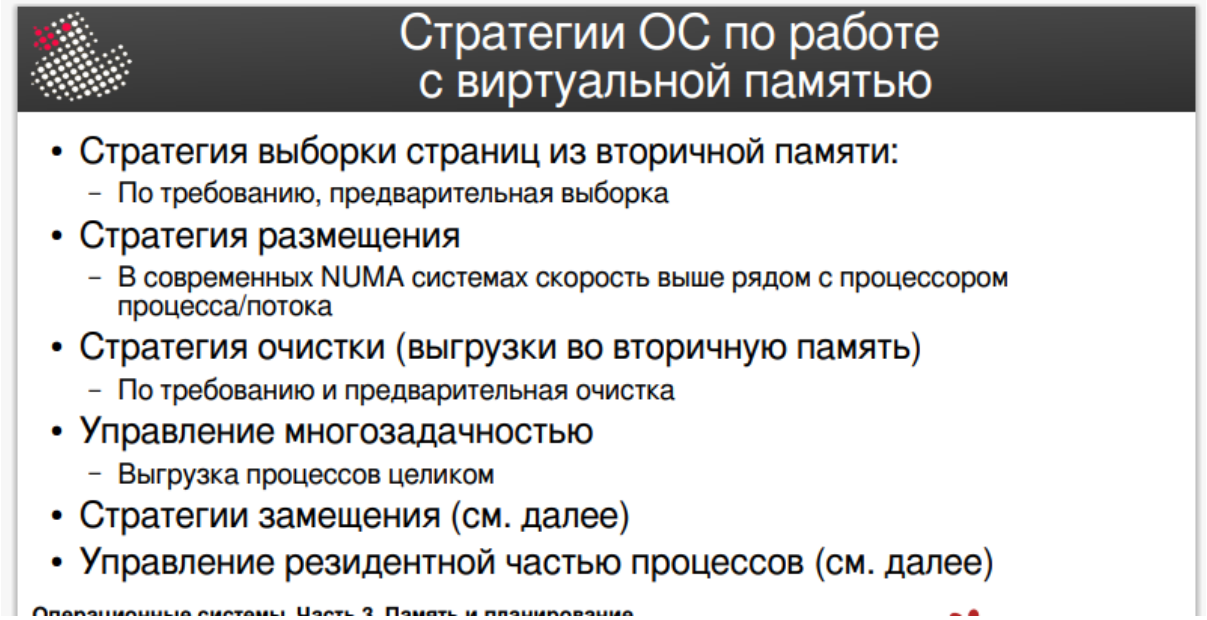


Рис. 8.12. Трансляция адреса при совместном использовании сегментации и страничной организации

47. Влияние размера страницы виртуальной памяти на ОС. Стратегии ОС по работе с виртуальной памятью.





48. Стратегии замещения страниц ОС. Часовой Алгоритм. Управление резидентной частью процесса.

Какие именно кадры работающего процесса нужно выбрать для замещения и использования другими процессами?

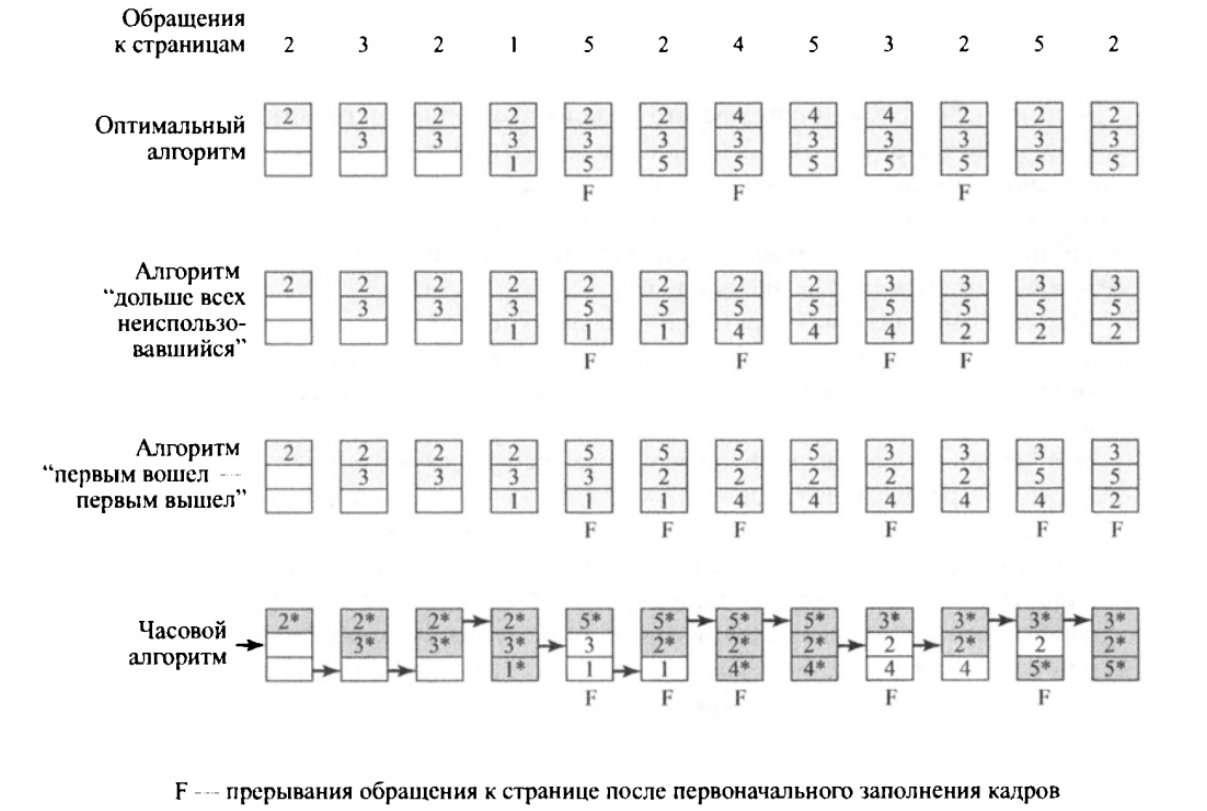
– Выгрузим кадр, обращения к которому не последует в ближайшее время

– Используется статистика прошлого поведения

– Необходимо учитывать locked frames

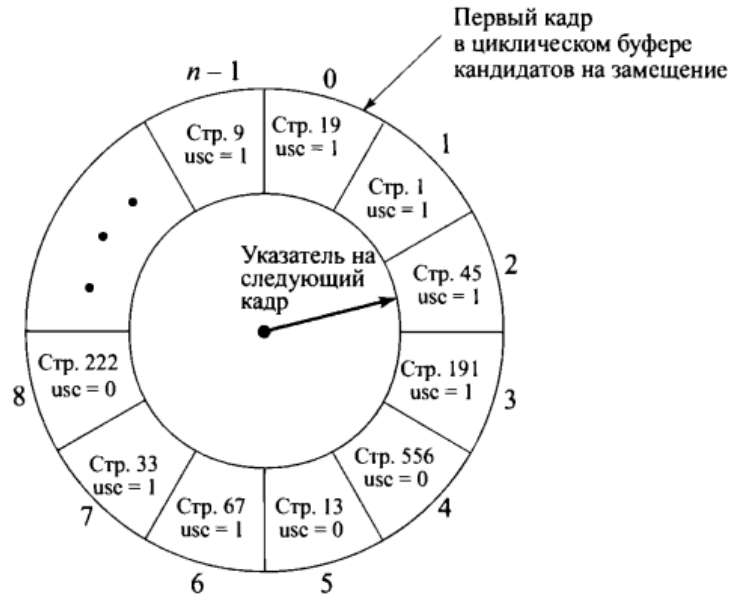
– не все страницы можно выгрузить (часть ядра, буферы, ...)

– Если кадр совместно используется много раз (н-р код разделяемой библиотеки) то его выгрузка может повлиять на много процессов

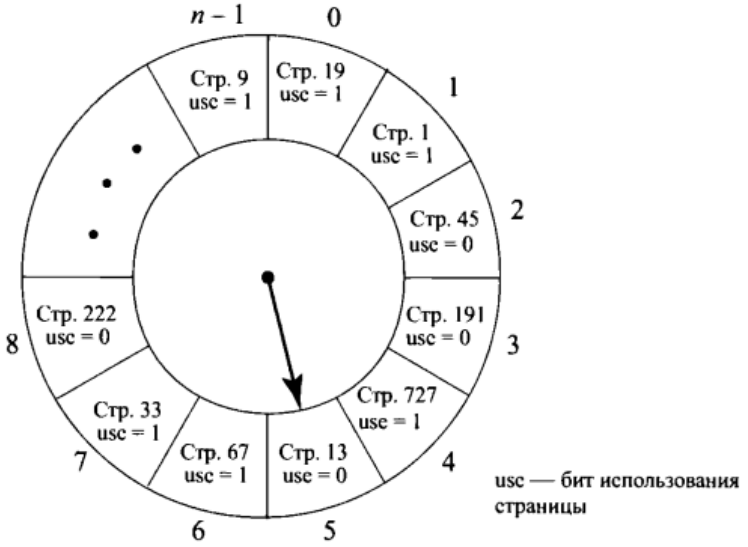


F — прерывания обращения к странице после первоначального заполнения кадров

Рис. 8.14. Поведение четырех алгоритмов замещения страниц



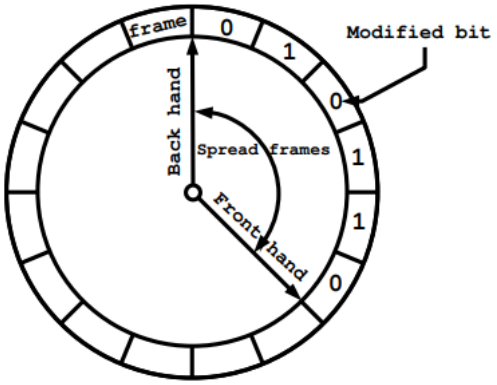
а) Состояние буфера непосредственно перед замещением страницы



б) Состояние буфера непосредственно после замещения страницы

Рис. 8.15. Пример работы часового алгоритма

Модифицированный Clock Algorith (Solaris 2.0 - 11)



- Алгоритм:
  - FH — сбрасывает бит модификации
  - Модификация фрейма устанавливает
  - BH — Проверяет
- Расстояние между стрелками меняется динамически
- На современных размерах памяти занимает много времени
  - 16Гб/4кб = 4194304 фреймов

Управление резидентной частью процессов

- Размер резидентной части:
  - При загрузке и работе процесса нет смысла держать в памяти все его страницы
  - Меньше памяти процессу → больше процессов в памяти
  - Меньше страниц процесса в памяти → больше pagefaults
  - После N-страниц дополнительное выделение памяти не приводит к кардинальному снижению pagefaults
  - Стратегии управления: Фиксированный и динамический размер
- Область видимости (scope)
  - Локальная — страница замещается у процесса вызвавшего pagefault
  - Глобальная — сканируются страницы всех процессов, кроме locked и сильно shared

Таблица 8.5. Управление резидентным множеством

	Локальное замещение	Глобальное замещение
Фиксированное распределение	• Количество кадров процесса фиксировано • Страница для замещения выбирается среди выделенных процессу кадров	• Невозможно
Переменное распределение	• Количество выделенных процессу кадров может время от времени изменяться • Страница для замещения выбирается среди выделенных процессу кадров	• Страница для замещения выбирается среди всех доступных кадров в основной памяти; это приводит к изменению размера резидентного множества процесса

49. Виды планирования процессов. Критерии краткосрочного планирования. Приоритеты.

Таблица 9.1. Типы планирования

Долгосрочное планирование	Решение о добавлении процесса в пул выполняемых процессов
Среднесрочное планирование	Решение о добавлении процесса к числу процессов, полностью или частично размещенных в основной памяти
Краткосрочное планирование	Решение о том, какой из доступных процессов будет выполняться процессором
Планирование ввода-вывода	Решение о том, какой из запросов процессов на операции ввода-вывода будет обработан доступным устройством ввода-вывода

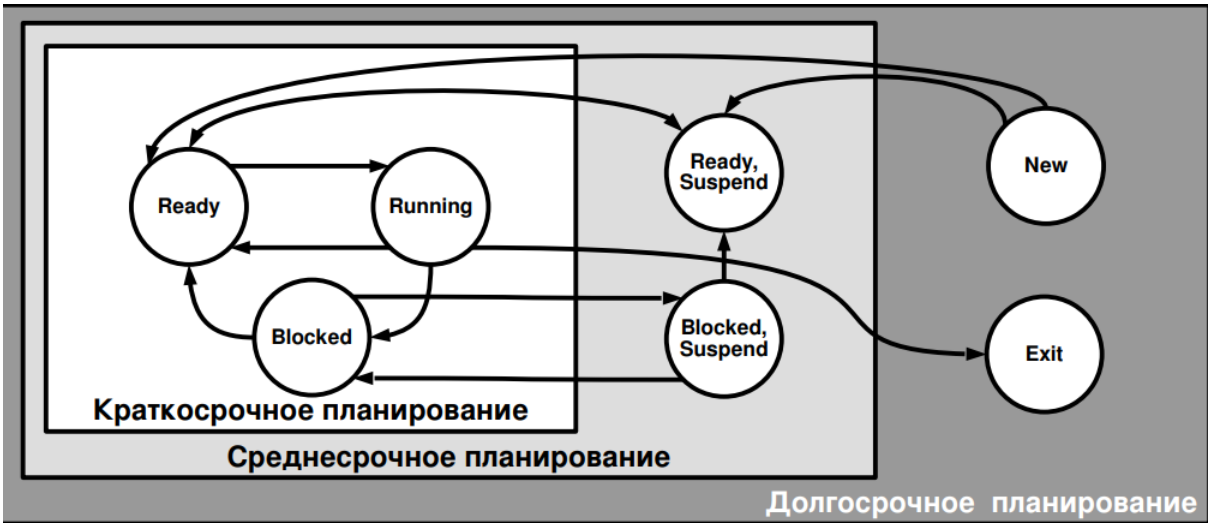


Таблица 9.2. Критерии планирования

Пользовательские, связанные с производительностью
Время оборота Интервал времени между передачей процесса для выполнения и его завершением. Включает время выполнения, а также время, затраченное на ожидание ресурсов, в том числе процессора. Критерий вполне применим для пакетных заданий
Время отклика В интерактивных процессах это время, истекшее между подачей запроса и началом получения ответа на него. Зачастую процесс может начать вывод информации пользователю, еще не окончив полной обработки запроса, так что описанный критерий — наиболее подходящий с точки зрения пользователя. Стратегия планирования должна пытаться сократить время получения ответа при максимизации количества интерактивных пользователей, время отклика для которых не выходит за заданные пределы
Предельный срок При указании предельного срока завершения процесса планирование должно подчинить ему все прочие цели максимизации количества процессов, завершающихся в срок

Окончание табл. 9.2

Пользовательские, иные
Предсказуемость Данное задание должно выполняться примерно за одно и то же количество времени и с одной и той же стоимостью, независимо от загрузки системы. Большие вариации времени выполнения или времени отклика дезориентируют пользователей. Это явление может сигнализировать о больших колебаниях загрузки или о необходимости дополнительной настройки системы для устранения нестабильности ее работы
Системные, связанные с производительностью
Пропускная способность Стратегия планирования должна пытаться максимизировать количество процессов, завершающихся за единицу времени, что является мерой количества выполненной системой работы. Очевидно, что эта величина зависит от средней продолжительности процесса; однако на нее влияет и используемая стратегия планирования
Использование процессора Этот показатель представляет собой процент времени, в течение которого процессор оказывается занятым. Для дорогих совместно используемых систем этот критерий достаточно важен; в однопользовательских же и некоторых других системах (типа систем реального времени) — менее важен по сравнению с рядом других
Системные, иные
Беспристрастность При отсутствии дополнительных указаний от пользователя или системы все процессы должны рассматриваться как равнозначные и ни один процесс не должен подвергнуться голоданию
Использование приоритетов Если процессам назначены приоритеты, стратегия планирования должна отдавать предпочтение процессам с более высоким приоритетом
Баланс ресурсов Стратегия планирования должна поддерживать занятость системных ресурсов. Предпочтение должно быть отдано процессу, который недостаточно использует важные ресурсы. Этот критерий включает использование долгосрочного и среднесрочного планирования

50. Использование приоритетов.

Использование приоритетов

- Процессам присваивается цифровой приоритет
  - В разных ОС - разные схемы назначения приоритетов
- Из очередей выбирается процесс с наивысшим приоритетом
- Если приоритеты процессов совпадают, то используется дополнительная стратегия
- Процессы с низким приоритетом могут голодать
- Приоритеты могут динамически изменяться

При выборе процесса планировщик начинает с очереди процессов с наивысшим приоритетом (RQ0). Если в очереди имеется один или несколько процессов, процесс для работы выбирается с использованием некоторой стратегии планирования. Если очередь RQ0 пуста, рассматривается очередь RQ1 и т.д.

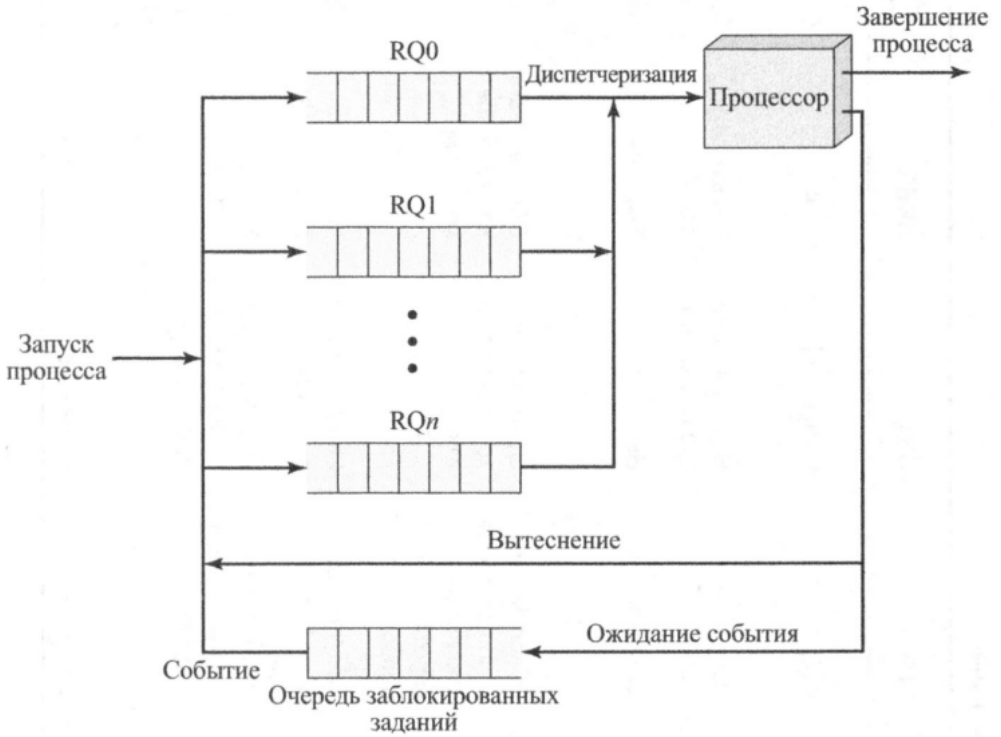


Рис. 9.4. Планирование с учетом приоритета процессов

Одна из основных проблем в такой чисто приоритетной схеме планирования состоит в том, что процессы с низким приоритетом могут оказаться в состоянии голодания. Это будет происходить при постоянном поступлении новых готовых к выполнению процессов с высоким приоритетом. Если такое поведение нежелательно, приоритет процесса может изменяться с его "возрастом" или историей выполнения (пример такой стратегии планирования будет приведен ниже).

51. Стратегии планирования FCFS, RR, SPN, SRT, HRRN, Feedback.

Стратегии планирования

- First Come First Served (FCFS) — аналог FIFO
- Round Robin — карусельное планирование
- Shortest Process Next — Короткие процессы вперед
- Shortest Remaining Time — Наименьшее время до завершения
- Highest Response Ratio Next — Наивысшее отношение отклика
- Feedback — снижения приоритета в зависимости от длительности времени исполнения
  - в случае blocked или preempted — feed it back

Сравнение стратегий

	FCFS	RR	SPN	SRT	HRRN	Feedback
Функция выбора	max(w)	Const TQ	min(s)	min(s-e)	max(w+s)/s	w↑ → prio↑ e↑ →prio↓
Preemption	No	At Time Quantum	No	At arrival to ready queue	No	At Time quantum
Throughtput	-	Can be low if TQ low	High	High	High	-
Responce time	Can be high	Good for short	Good for short	Good	Good	-
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect	Bad for short and high I/O	fair	Bad for long	Bad for long	Balanced	Can preffer high IO proc.
Starvation	No	No	Possible	Possible	No	Possible

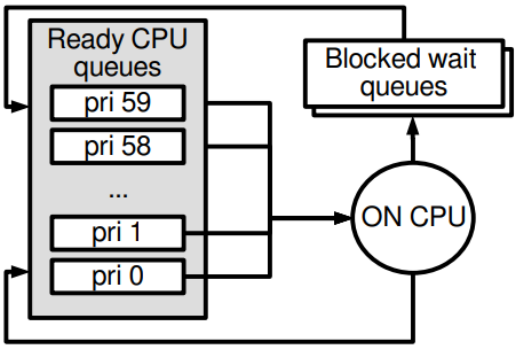
w — общее время ожидания процесса в очередях, e — общее время выполнения,
s — общее время, предполагаемое или заданное, для обслуживания процесса, включая e

Таблица 9.3. Характеристики различных стратегий планирования

	FSCF	Круговая	SPN	SRT	HRRN	Со снижением приоритета
Функция выбора	$\max[w]$	const	$\min[s]$	$\min[s-e]$	$\max\left(\frac{w+s}{s}\right)$	См. текст
Режим решения	Невытесняющий	Вытесняющий (по времени)	Невытесняющий	Вытесняющий (по решению)	Невытесняющий	Вытесняющий (по времени)
Пропускная способность	Не важна	Может быть низкой при малом кванте времени	Высокая	Высокая	Высокая	Не важна
Время отклика	Может быть большим, в особенности при больших отклонениях во времени выполнения процесса	Обеспечивает хорошее время отклика для коротких процессов	Обеспечивает хорошее время отклика для коротких процессов	Обеспечивает хорошее время отклика	Обеспечивает хорошее время отклика	Не важна
Накладные расходы	Минимальны	Минимальны	Могут быть высокими	Могут быть высокими	Могут быть высокими	Могут быть высокими
Влияние на процессы	Плохо сказывается на коротких процессах и процессах с интенсивным вводом-выводом	Беспристрастна	Плохо сказывается на длинных процессах	Плохо сказывается на длинных процессах	Хороший баланс	Может привести к предпочтению процессов с интенсивным вводом-выводом
Голодание	Отсутствует	Отсутствует	Возможно	Возможно	Отсутствует	Возможно

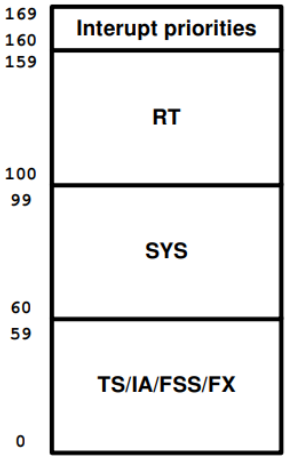
52. cpo. Feedback планировщик и классы планирования ОС UNIX SVR4.

globpri	quantum	tqexp	slpret	maxwait	lwait
59	2	49	59	32000000	59
58	4	48	58	0	59
57	4	47	58	0	59
...					
40	4	30	55	0	55
...					
30	8	20	53	0	53
29	12	19	52	0	52
...					
21	12	11	52	0	52
20	12	10	52	0	52
19	16	9	51	0	51
...					
12	16	2	51	0	51
11	16	1	51	0	51
10	16	0	51	0	51
9	20	0	50	0	50
...					
1	20	0	50	0	50
0	20	0	50	0	50



globpri - глобальный приоритет,
quantum - выделяемый квант времени (мс),
tqexp - приоритет, если квант полностью выбран CPU,
slpret - приоритет, если ожидаем ресурса
maxwait — граничное время длинного ожидания (мс)
lwait - прирритет, в случае длинного ожидания ресурса

Классы планирования SVR4



- TimeSharing — разделение времени (Feedback)
- InterActive — интерактивный (boost к активному приложению)
- Fixed, System, RealTime — классы с фиксированными приоритетами
- Fair Share Scheduler — справедливый планировщик
- Отдельные приоритеты для Interrupt threads
- Учет афинити для NUMA

Класс приоритета	Глобальное значение	Последовательность планирования
Реальное время	159 • • • • 100	Первые ↓
Ядро	99 • • 60	↓
Разделение времени	59 • • • • 0	↓ Последние

Рис. 10.12. Очереди диспетчера SVR4

53. Справедливое планирование.

Планирование осуществляется исходя из приоритетов с учетом приоритета процесса, недавнего использования им процессора и недавнего использования процессора группой, к которой он принадлежит. Чем больше числовое значение приоритета, тем ниже сам приоритет. Для процесса j из группы k применимы следующие формулы:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2}$$

$$P_j(i) = \text{Base}_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4W_k}$$

где

$CPU_j(i)$ — мера загруженности процессора процессом j на интервале i;

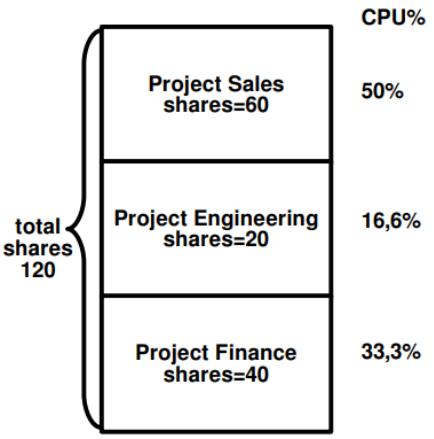
$GCPU_k(i)$ — мера загруженности процессора группой k на интервале i;

$P_j(i)$ — приоритет процесса j в начале интервала i (меньшее значение соответствует большему приоритету);

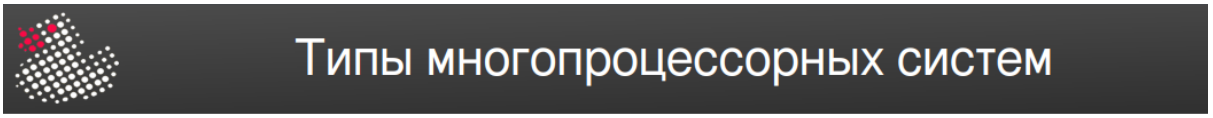
Base_j — базовый приоритет процесса j;

W_k — вес, назначенный группе k ($0 < W_k \le 1$ и $\sum_k W_k = 1$).

- Выделяет время процессора на основании заданных user shares
- Используется как замена TS/IA классов
- Учитывает все процессоры в системе



54. Планирование в многопроцессорных системах. Типы многопроцессорных систем с точки зрения организации планирования. Гранулярность и проектирование планировщиков процессов и потоков для многопроцессорных систем.



- Слабосвязанные (распределенные, кластеры)
  - Набор систем со своей основной памятью и подсистемой ввода-вывода
  - Распределение заданий между системами
  - Распределение общих данных и результатов
- Функционально-специализированные
  - Ведущий процессор выполняет координацию
  - Ведомые процессоры выполняют вычисления
- Сильносвязанные процессоры
  - Имеют общую память и систему кэшей
  - Управляются одной ОС

- Гранулярность синхнхронизации — частота синхронизации между процессами в системе
- Fine (тонкая) — параллельные вычисления на уровне отдельных машинных команд
  - Менее 20 команд
- Medium (средняя) — на уровне одного приложения
  - 20-200 команд
- Coarse (губая) — на уровне взаимодействующих процессов
  - 200-2000 команд
- Very Coarse (очень грубая) — на уровне распределенных систем
  - 2000-1 000 000
- Independent (независимая) — нет синхронизации, независимые процессы

Вопросы проектирования планировщиков в случае (сильно) многопроцессорных систем

- Назначение процессов процессорам
  - Статическое — выделяем процессор для процесса
  - Динамическое — общая очередь для всех процессов
  - Динамическая балансировка нагрузки
- Использование многозадачности на отдельных процессорах
  - Нужна ли для статического назначения многозадачность?
- Диспетчеризация процесса
  - Так ли плох будет FCFS? Влияние выбора алгоритма диспетчеризации снижается.

Подходы к планированию потоков

- Load Sharing — глобальная очередь потоков
  - Равномерное распределение нагрузки
  - Нет централизованного планировщика
  - Минусы — блокировки центральной очереди, промахи кэшей, неэффективность при тонкой средней гранулярности
- Gang scheduling — связанные потоки распределяются на связанные процессы по одному на процессор
  - Снижение накладных расходов на планирование при тонкой и средней гранулярности
- Dedicated processor assignment — назначается пул процессоров равный количеству потоков
  - В экстремальном случае увеличивает простои процессора
  - Полное устранение переключений повышает скорость работы
- Динамическое планирование
  - В отсутствии свободных CPU ресурсы изымаются из процесса, использующего несколько CPU

55. ОС реального времени и планировщики. Deadline-планирование.

Планирование реального времени

Планирование реального времени является одной из областей, в которых ведутся активные исследовательские работы. В этом подразделе мы приведем краткий обзор различных подходов к проблеме планирования реального времени и рассмотрим два распространенных класса алгоритмов планирования.

В обзоре алгоритмов планирования реального времени [196] замечено, что различные алгоритмы планирования зависят от того, 1) выполняет ли система анализ планируемости, и если выполняет, то 2) как именно, статически или динамически, и 3) к чему он приводит — к непосредственному выполнению функции планирования или же к построению расписания, согласно которому в процессе работы осуществляется диспетчеризация заданий. На основе этого автор определяет следующие классы алгоритмов.

- **Статическое планирование с использованием таблиц.** При этом выполняется статический анализ осуществимости планирования; результатом анализа является план, который в процессе работы системы определяет, когда должно начаться выполнение заданий.
- **Статическое вытесняющее планирование на основе приоритетов.** В этом случае также выполняется статический анализ, но расписание не создается. Вместо этого на основе проведенного анализа заданиям назначаются приоритеты, с тем чтобы далее можно было использовать традиционный вытесняющий планировщик, работающий с учетом приоритетов заданий.
- **Динамическое планирование на основе расписания.** Осуществимость планирования определяется не статически, а динамически, в процессе выполнения.

Поступающее в систему задание принимается только в том случае, если определена возможность его выполнения с учетом всех временных требований. Одним из результатов анализа является расписание, используемое для принятия решения о диспетчеризации задания.

- **Динамическое планирование наилучшего результата**. При этом анализ осуществимости планирования не выполняется; система пытается удовлетворить все предельные сроки и снимает те выполняющиеся процессы, предельные сроки которых нарушены.

Статическое планирование с использованием таблиц применимо для периодических заданий. Входной информацией для анализа являются время поступления заданий в систему, время выполнения, предельные сроки выполнения и относительный приоритет каждого задания. Планировщик пытается разработать такой план работы, который удовлетворял бы всем временным требованиям заданий. Такой подход является предсказуемым, но абсолютно не гибким, поскольку любое изменение требований любого задания приводит к необходимости пересмотра всего расписания. Типичными представителями этой категории алгоритмов планирования являются планирование наиболее раннего предельного срока и другие, рассматриваемые далее, алгоритмы планирования периодических заданий.

Статическое вытесняющее планирование на основе приоритетов использует тот же вытесняющий механизм планирования с приоритетами, что и большинство обычных многозадачных операционных систем. В таких системах для определения приоритетов могут использоваться самые различные факторы. Например, в системе с разделением времени приоритет процесса изменяется в зависимости от того, на что ориентирован этот процесс — на вычисления или на операции ввода-вывода. В системах реального времени назначение приоритетов связано с временными ограничениями каждого из заданий. Примером такого подхода может служить частотно-монотонное планирование, рассматриваемое ниже в данной главе, которое назначает заданиям статические приоритеты на основе информации об их периодах.

В случае использования **динамического планирования на основе расписания** после поступления задания в систему (но до начала его выполнения) предпринимается попытка создать расписание, которое содержит как все имеющиеся в системе задания, так и вновь поступившее. Если удается создать такое расписание, что при выполнении удовлетворяются временные ограничения как нового задания, так и уже имеющихся в системе заданий, оно принимается планировщиком.

Динамическое планирование наилучшего результата используется во многих современных коммерческих системах реального времени. При поступлении нового задания система назначает ему приоритет на основе характеристик этого задания. При этом подходе обычно используется планирование, учитывающее предельные сроки, — наподобие планирования наиболее раннего предельного срока. Как правило, поступающие задания непериодические, а потому статический анализ планирования неприменим. При таком типе планирования мы не знаем, будут ли удовлетворены временные ограничения задания до тех пор, пока задание не будет полностью выполнено (или пока не будут нарушены временные ограничения). Именно это и является основным недостатком данной схемы; преимущество же динамического планирования наилучшего результата — в простоте реализации.

На рис. 10.8 показан простой пример RMS. Экземпляры заданий пронумерованы последовательно. Как можно видеть, для задания 3 второй экземпляр не выполняется, поскольку предельное время просрочено. Третий экземпляр испытывает вытеснение, но все равно остается в состоянии завершиться до предельного срока завершения.

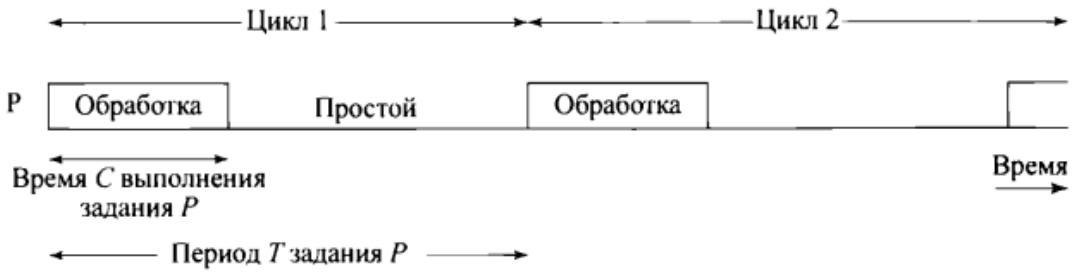
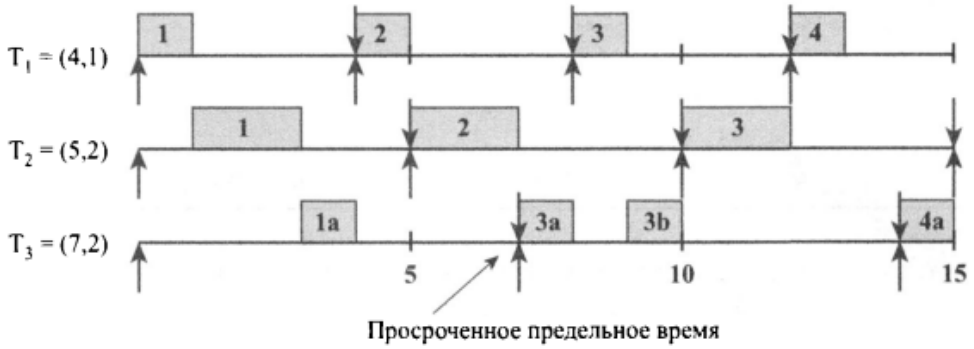


Рис. 10.7. Временная диаграмма периодического задания

а) Времена поступления и предельные времена завершения задания $T_i = (P_i, C_i)$
P_i — период, C_i — время обработки



б) Результаты планирования

Рис. 10.8. Пример работы RMS

Одной из характеристик эффективности алгоритма периодического планирования является его гарантия соответствия всем жестким предельным срокам. Предположим, что у нас имеется n заданий, каждое из которых имеет свое фиксированное время выполнения и период. Тогда необходимым условием соответствия всем жестким предельным срокам является выполнение следующего неравенства:

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\ldots+\frac{C_n}{T_n}\le 1 \qquad (10.1)$$

Сумма загруженности процессора разными заданиями не может превышать 1, что соответствует полной загрузке процессора. Неравенство (10.1) определяет верхнюю границу количества заданий, которые может успешно обслуживать идеальный алгоритм планирования. Для конкретного реального алгоритма граница может оказаться ниже. Так, можно показать, что для алгоритма RMS справедливо следующее неравенство:

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\ldots+\frac{C_n}{T_n}\le n(2^{1/n}-1) \qquad (10.2)$$

В табл. 10.5 приведены некоторые значения верхней границы для метода RMS. По мере возрастания количества заданий верхняя граница стремится к значению $\ln 2 \approx 0,693$.

Таблица 10.5. Значения верхней границы загруженности для метода RMS

n	$n(2^{1/n}-1)$
1	1,000
2	0,828
3	0,779
4	0,756
5	0,743
6	0,734
.	.
.	.
.	.
∞	$\ln 2 \approx 0,693$

В качестве примера рассмотрим три периодических задания (здесь $U_i = C_i/T_i$).

- *Задание* P_1: $C_1 = 20$; $T_1 = 100$; $U_1 = 0,2$.
- *Задание* P_2: $C_2 = 40$; $T_2 = 150$; $U_2 = 0,267$.
- *Задание* P_3: $C_3 = 100$; $T_3 = 350$; $U_3 = 0,286$.

Общая загруженность процессора этими тремя заданиями составляет

$$0,2 + 0,267 + 0,286 = 0,753.$$

Верхняя граница загруженности этих трех задач при использовании метода RMS составляет

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\frac{C_3}{T_3}\le 3(2^{1/3}-1)=0,779.$$

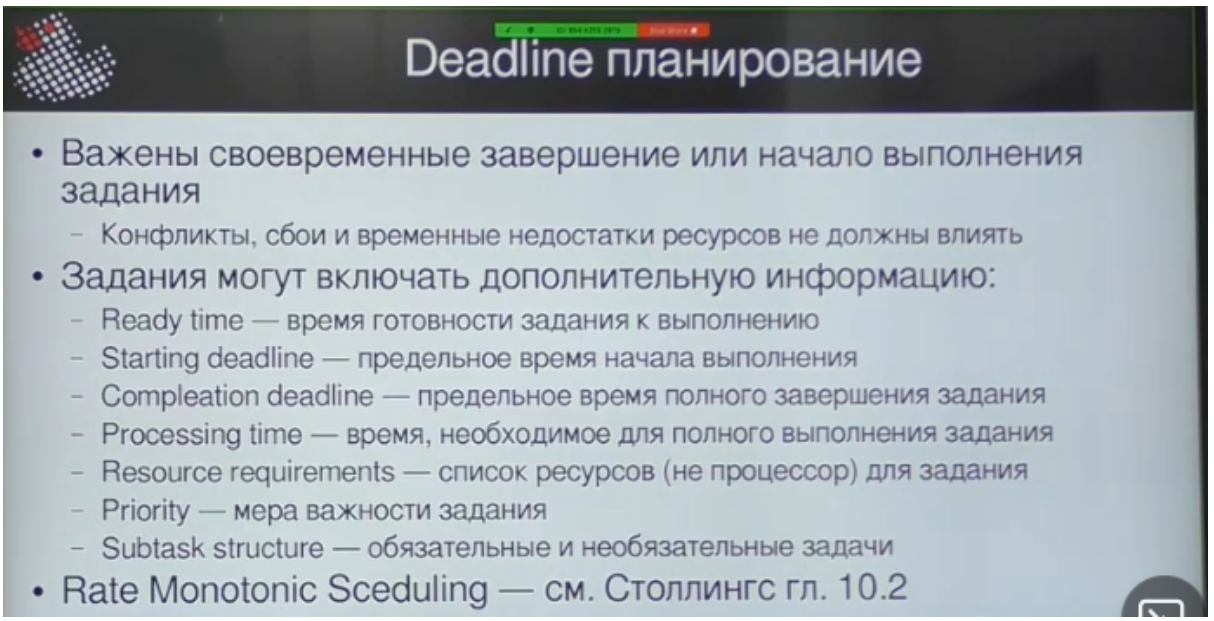
Поскольку общая загруженность процессора по обработке приведенных заданий ниже верхней границы для метода RMS, можно сделать вывод,

что при RMS-планировании будут успешно выполнены все задания. Можно также показать, что верхняя граница из (10.1) справедлива для метода наиболее раннего предельного срока. Таким образом, при применении планирования с наиболее ранним предельным сроком можно достичь более высокой загрузки процессора и, соответственно, обработать большее количество заданий. Тем не менее метод RMS широко распространен и используется во многих промышленных приложениях. В работе [221] это поясняется следующими причинами.

1. На практике отличие производительности невелико. Кроме того, неравенство (10.2) консервативно, и на практике зачасгую достигается 90%-ная загрузка процессора.

2. Большинство жестких систем реального времени содержат мягкие компоненты, такие как некритичный вывод на экран или встроенное самотестирование, выполняющееся с низким приоритетом. Эти компоненты используют процессорное время, которое остается после RMS-планирования жестких заданий.

3. При использовании RMS проще обеспечить стабильность. Когда система не в состоянии обеспечить все предельные сроки в силу перегруженности или временных ошибок, необходимо гарантировать выполнение предельных сроков для подмножества обязательных заданий. При статическом назначении приоритетов гарантируется только корректность выполнения основных задач с относительно высокими приоритетами. Это может быть достигнуто и в RMS-планировании путем реструктурирования обязательных задач для повышения их частоты либо посредством изменения приоритетов RMS для подмножества обязательных задач. При планировании с наиболее ранним предельным сроком приоритеты периодических заданий изменяются от одного периода к другому, что усложняет обеспечение корректной работы обязательных заданий с жесткими предельными сроками.


Deadline планирование
• Важены своевременные завершение или начало выполнения задания
– Конфликты, сбои и временные недостатки ресурсов не должны влиять
• Задания могут включать дополнительную информацию:
– Ready time — время готовности задания к выполнению
– Starting deadline — предельное время начала выполнения
– Compleation deadline — предельное время полного завершения задания
– Processing time — время, необходимое для полного выполнения задания
– Resource requirements — список ресурсов (не процессор) для задания
– Priority — мера важности задания
– Subtask structure — обязательные и необязательные задачи
• Rate Monotonic Sceduling — см. Столлингс гл. 10.2

56. Проблема инверсии приоритетов, типы инверсии и способы решения в планировщике.

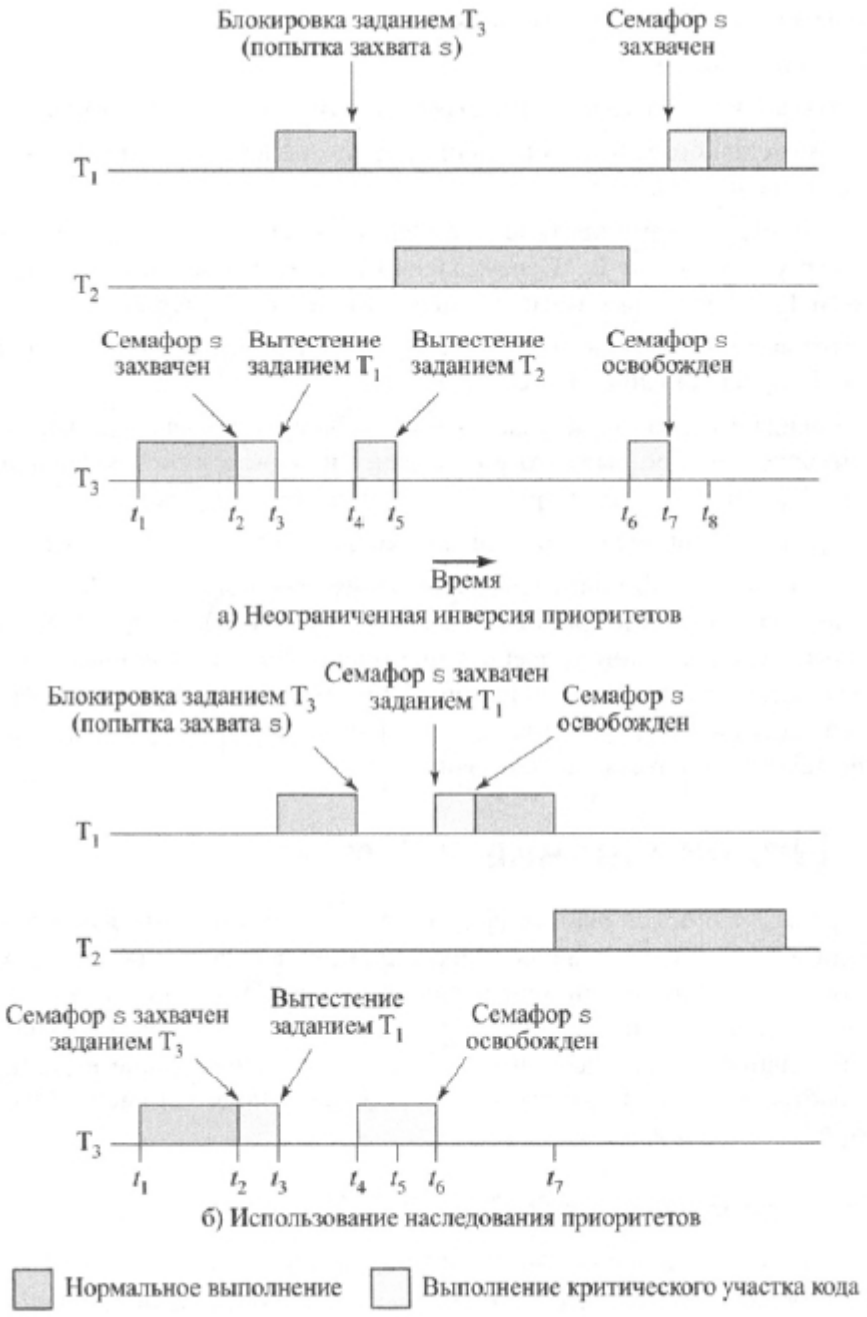


Рис. 10.9. Инверсия приоритетов

Основная идея **наследования приоритетов** (priority inheritance) заключается в том, что задача с более низким приоритетом наследует приоритет любой высокоприоритетной задачи, ожидающей совместно используемый ресурс. Это изменение приоритета происходит, как только более высокоприоритетное задание блокируется в ожидании ресурса; оно должно заканчиваться, когда ресурс освобождается задачей с более низким приоритетом. На рис. 10.9, *б* показано, что наследование приоритетов решает проблему неограниченной инверсии приоритетов, показанной на рис. 10.9, *а*. Соответствующий порядок событий оказывается следующим.

t_1: T_3 начинает выполняться.

t_2: T_3 захватывает семафор `s` и входит в критический участок кода.

t_3: T_1, имеющий более высокий приоритет, чем приоритет T_3, вытесняет T_3 и начинает выполняться.

t_4: T_1 пытается войти в критический участок кода, но блокируется, поскольку семафор уже захвачен T_3. T_3 немедленно (временно) получает тот же приоритет, что и T_1. T_3 продолжает выполнение критического участка.

t_5: T_2 готов к выполнению, но поскольку T_3 теперь имеет более высокий приоритет, T_2 не в состоянии вытеснить T_3.

t_6: T_3 покидает критический участок и разблокирует семафор: уровень его приоритета опускается до бывшего у него ранее приоритета по умолчанию. T_1 вытесняет T_3, блокирует семафор и входит в критический участок.

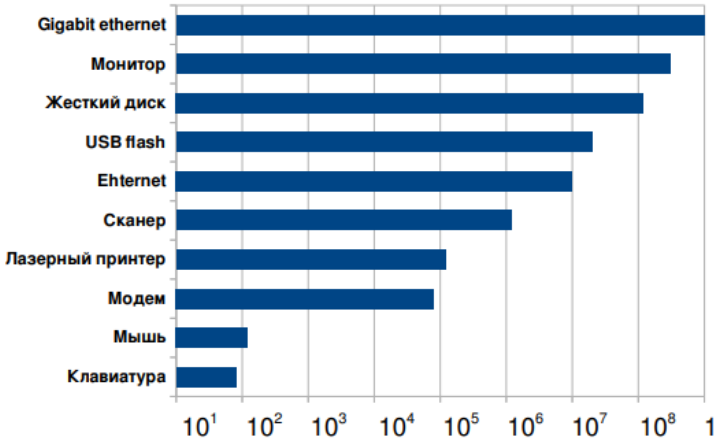
t_7: T_1 приостанавливается по причине, не связанной с T_2, и T_2 начинает выполнение.

Именно этот подход был применен для решения проблемы Pathfinder.

В подходе с **потолком приоритета** с каждым ресурсом связан приоритет. Приоритет, назначенный ресурсу, на один уровень выше приоритета его наиболее приоритетного пользователя. Затем планировщик динамически назначает этот приоритет любой задаче, которая обращается к ресурсу. Когда задание заканчивает работу с ресурсом, его приоритет возвращается в нормальное состояние.

.57. Ввод-вывод. Современные устройства и скорости обмена, развитие способов ввода-вывода, логическая структура ввода-вывода.

Устройства ввода-вывода

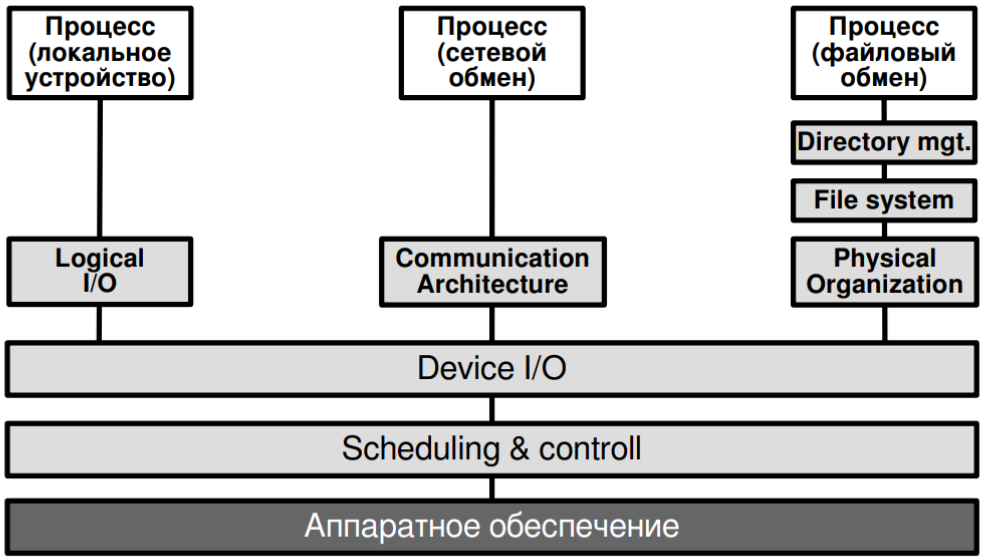


- Устройства:
  - Пользовательские
  - Внутренние
  - Связи
- Характеристики:
  - Скорость
  - Применение
  - Сложность управления
  - Единица передачи
  - Представление данных
  - Обстоятельства ошибок

Развитие ввода вывода

- Программируемый ввод-вывод
  - Процессор непосредственно управляет периферийным устройством через его шину и регистры ...
  - ... или контроллер, который подключается к шине и имеет набор управляющих регистров
- Ввод-вывод с использованием прерываний
  - В контроллере добавляются прерывания, исключая ожидания
- Прямой доступ к памяти (Direct Memory Access)
  - В контроллере добавляются регистры и счетчики для обеспечения переноса области буфера в контроллере в область памяти
  - Контроллер превращается в отдельный вычислительный модуль (канал ввода-вывода) с процессором и системой команд
  - В канал ввода-вывода добавляется доп. оборудование и микропрограммы и контроллер становится отдельным вычислительным устройством полностью берущим на себя управление вводом-выводом с группой устройств (процессор ввода-вывода)

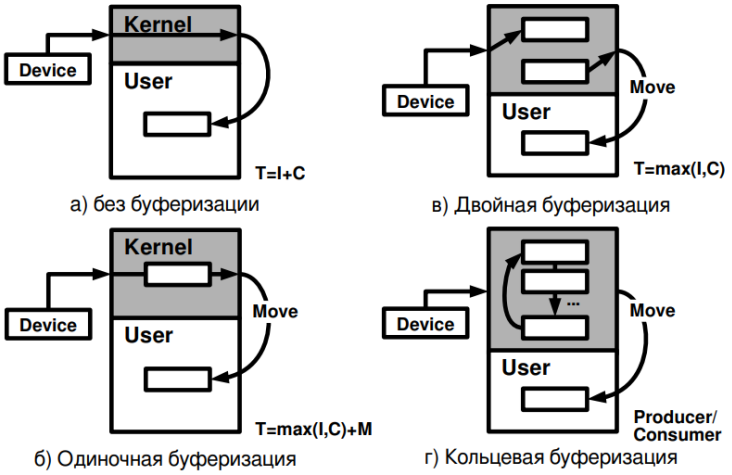
Логическая структура ввода-вывода



- **Логический ввод-вывод**. Модуль логического ввода-вывода обращается с устройством как с логическим ресурсом и не обращает внимания на детали фактического управления устройством. Логический модуль ввода-вывода работает посредником между пользовательскими процессами и устройством, позволяя им работать с последним с использованием идентификатора устройства и простых команд, таких как открытие, закрытие, чтение и запись.
- **Устройство ввода-вывода**. Запрошенные операции и данные (буферизированные символы, записи и т.п.) конвертируются в соответствующие последовательности инструкций ввода-вывода, команд управления каналом и команд контроллера. Для более эффективного использования устройства может быть применена буферизация.
- **Планирование и контроль**. На этом уровне происходят реальная организация очередей и планирование операций ввода-вывода, а также управление выполнением операций. Таким образом, на этом уровне осуществляются работа с прерываниями, получение и передача информации о состоянии устройства. Это уровень программного обеспечения, которое непосредственно взаимодействует с контроллером ввода-вывода, а следовательно, с аппаратным обеспечением устройства.

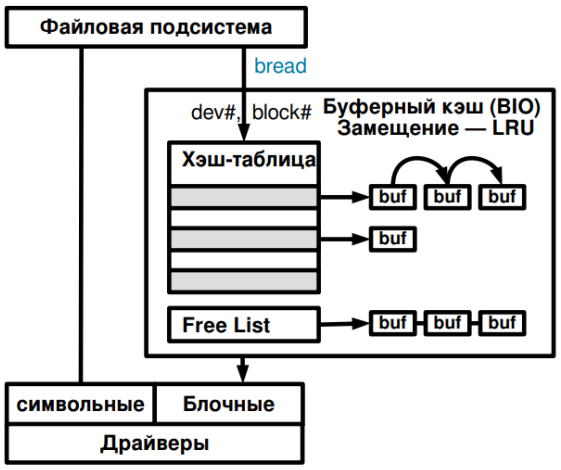
58. Буферизация ввода вывода. Ввод-вывод в UNIX SVR4.

Буферизация ввода-вывода



- Устройства:
  - Блочные
  - Символьные (потоковые)
- Легенда:
  - I — время ввода-вывода
  - C — время обработки
  - M — время пересылки

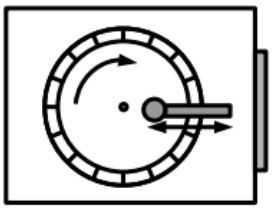
Ввод-вывод в SVR4



- Два вида:
  - Буферизированный ввод-вывод (через bio)
  - Небуферизированный ввод-вывод (через DMA)
- Для медленных устройств (терминалы...)
  - Используются отдельные символьные буферы

59. Диски и дисковое планирование.

Диски

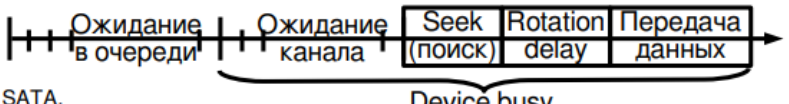


Hard Disk

IDE, SATA, SCSI, SAS interfaces

Solid State Disk

SATA, SAS, NVMe interfaces



$$T_д = T_о + T_к + T_п + \frac{1}{2r} + \frac{b}{rN}$$

- Тд — время доступа
- То — время ожидания в очереди ОС
- Тк — время освобождения канала
- Тп — время поиска
- r — скорость вращения RPM
- N — количество байт на дорожке
- b — количество байт в запросе

Дисковое планирование

- FIFO — «справедливый метод»
  - Все процессы получают одинаковый доступ к диску
  - Выгоден для небольшого потока запросов, на большом превращается в случайный доступ (издержки большие)
- PRI — на основе от приоритета процесса
  - Выгоден с точки зрения ОС (см. Feedback) для коротких заданий, длинным плохо.
- LIFO — Использует преимущества локальности данных
  - Хороша для транзакционных систем
- А если учитывать при планировании текущее состояние (н.р. - дорожка) диска?
- SSTF — Shortest Service Time First (Минимизация времени поиска)
- SCAN (elevator algorithm) — Ездим туда-сюда по диску и обслуживаем запросы
  - Предпочитает центр диска и плохо «относится» к запросам для только что пройденных дорожек
- C-SCAN — Ездим по диску во время операций в одну сторону, быстрый возврат
- N-step-SCAN — разделяет очередь на подочереди длинной N, 1 подочередь за 1 SCAN, если в подочереди запросов меньше N, то выполнить ее обработку в следующий SCAN.
- FCSCAN — две подочереди, пока одна обрабатывается, вторая заполняется.

60. Концепции RAID.

(Redundant Array of Independent Disks - избыточный массив независимых дисков).
1. RAID - это набор физических дисков, рассматриваемых операционной системой как единый логический диск.
2. Данные распределены по физическим дискам массива.
3. Избыточная емкость дисков используется для хранения контрольной информации, гарантирующей восстановление данных в случае отказа одного из дисков.

Благодаря большому количеству дисков повышается производительность, но увеличивается вероятность сбоя. Именно поэтому RAID предусматривает хранение информации для восстановления данных.

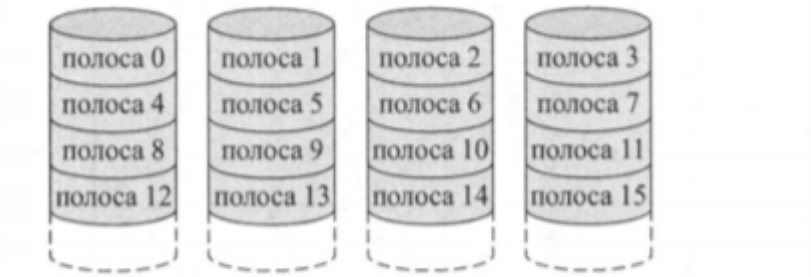
Несколько дисков вместо одного используются потому, что производительность вторичных запоминающих устройств растет медленнее, чем других компонентов, а несколько дисков позволяют выполнять несколько операций ввода-вывода одновременно.

Из семи уровней RAID чаще всего используются 0, 1, 5 и 6.

RAID 2 и 3 не используются из-за того, что они призваны использоваться в среде с большим количеством ошибок, в то время как диски обладают высокой надежностью.

..61. RAID-0, 1, 10, 0+1.

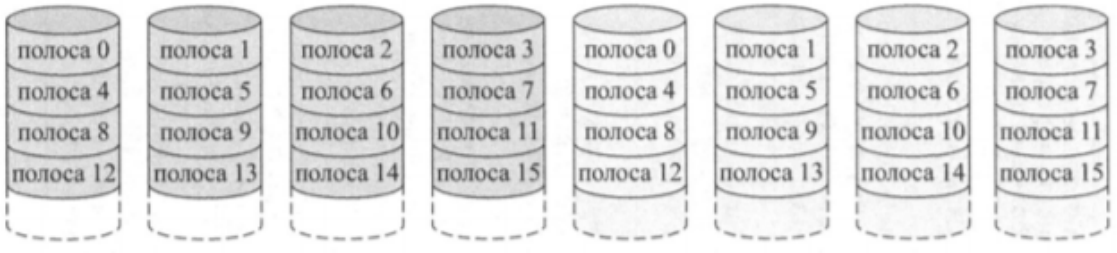
0 не является настоящим RAID - уровнем, поскольку никак не пользуется избыточностью. Основным плюсом RAID- 0 является очень высокая скорость ввода-вывода. Она достигается за счет того, что данные расщепляются(stripped) и записываются на все доступные диски сразу(по схеме 1 дорожка 1 диска,1 дорожка 2 диска,1 дорожка 3 диска,2 дорожка 1 диска,2 дорожка 2 диска и т.д.)



а) RAID0 (без избыточности)

Из-за такой записи повреждение одного диска приводит к повреждению всех данных.

1 зеркалирует данные (mirroring, создание зеркала).
Избыточность достигается не путем вычислений, а просто через дублирование данных. При этом используется такая же схема, как и при RAID-0.



б) RAID 1 (отражение)

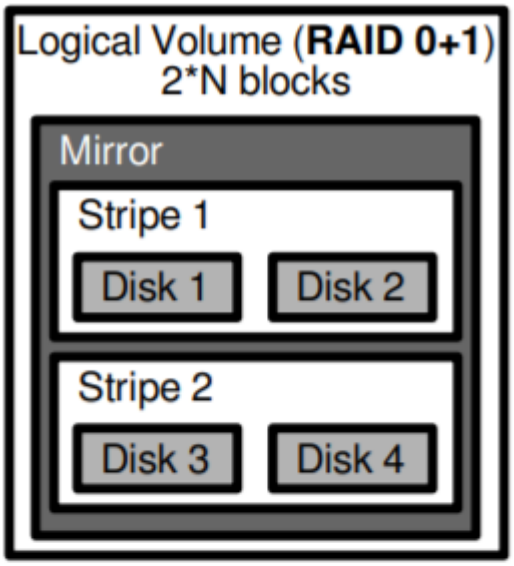
В качестве плюсов можно выделить:
простоту восстановления данных,
возможность выбирать между дисками (при обращении выбирается тот, у которого минимальная задержка из-за вращения и минимальное время поиска.

запись можно обновлять параллельно на обоих дисках(таким образом время определяется скоростью более медленной операции)
Минус - требуется в два раза больше дисков, поэтому RAID-1 используется только для самых важных файлов.

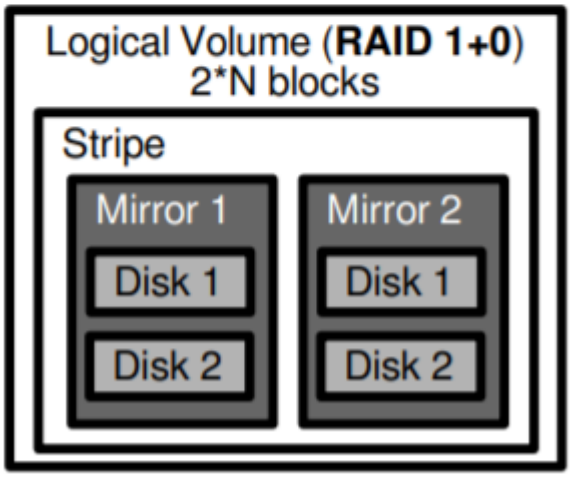
RAID-1 лучше подходит для чтения и может превышать скорость RAID-0 в два раза в среде, ориентированной на транзакции.

RAID - 1+0 =RAID 10
Массив RAID - 0, собранный из RAID-1

б)

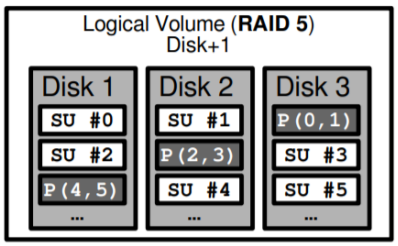


в)



62. RAID 4,5,6. Аппаратные дисковые массивы.

RAID -4 также почти не используется.



- Подсчитывает четность для SU
  - P(0,1) = SU(0)⊕SU(1)
  - Если сбой диска 2 SU(1)=P(0,1)⊕SU(0)
- Характеристики (a):
  - Высокая надежность, избыточность Disk+1
  - Дополнительные вычисления при записи, и при чтении в случае сбоя диска
  - Пропускная способность на чтение как у RAID 0 и меньше, чем одинарная на запись
  - Скорость обработки запросов для чтения как у RAID 0 и меньше, чем одинарная на запись
- RAID 4 — выделенный диск с четностью
- RAID 6 — двойная размазанная четность, по разным алгоритмам, избыточность Disk+2

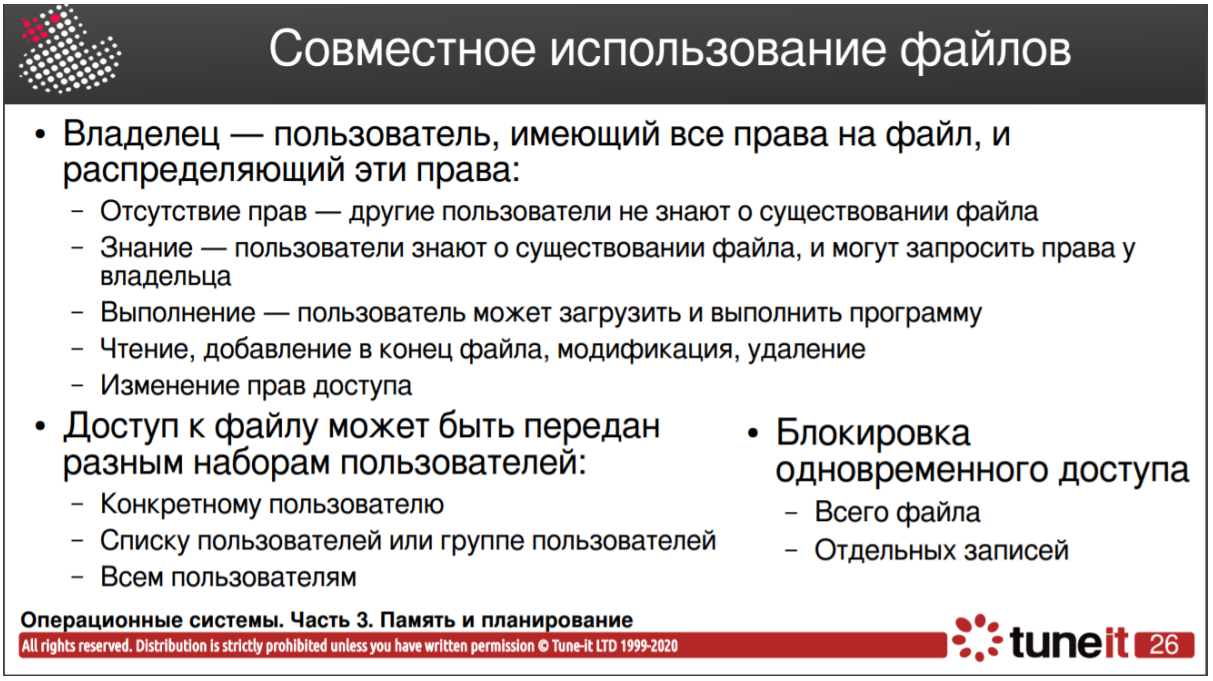
При уровне RAID 5 может быть восстановлен только один диск, при уровне RAID-6 - 2.

63. Файловый ввод-вывод, основные определения. Задачи ОС по управлению файлами. Совместное использование файлов.

Файл это коллекция данных со следующими свойствами:
– Наличие структуры (может быть сложной, может быть несколько в одном файле)
– Возможность долгосрочного существования
– Возможность совместного использования процессами
● Основные файловые операции
– Создание, удаление, открытие, закрытие, чтение, запись, выбор позиции, контрольные операции, блокирование

Поле — одиночный элемент записи, обычно одно значение.
Запись — набор полей
Файл — совокупность записей, относящихся к однородному набору данных.
База данных — файл со сложной взаимозависимой структурой полей и записей.
Операции, проводимые с файлом, могут влиять на его структуру
Файл может не иметь структуры (например, представлять собой поток символов)

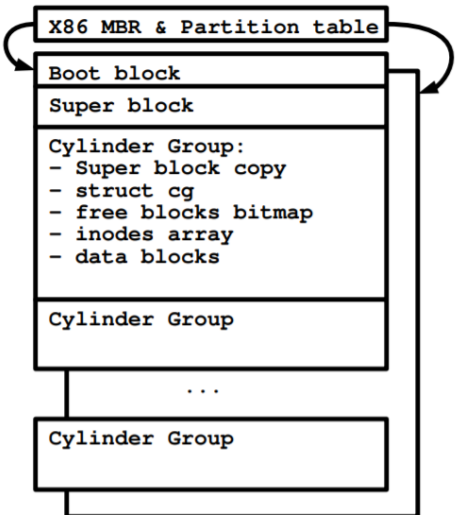
Задачи ОС по управлению файлами:
● Возможность хранить файлы и выполнять пользователю над ними операции: не
– Создание, удаление, чтение и изменения файлов
– Управление доступом к файлам для себя и других пользователей
– Перемещение данных между файлами
– Выполнение резервного копирования и восстановления файлов
– Обеспечение возможности работы с файлами по именам, удобным для пользователя
● Гарантия корректности данных
● Обеспечение приемлемой производительности
● Поддержка различных типов устройств хранения
● Минимизация или исключения потерь и повреждения данных
● Обеспечение базового набора функций ввода-вывода
● Обеспечение совместного использования файлов



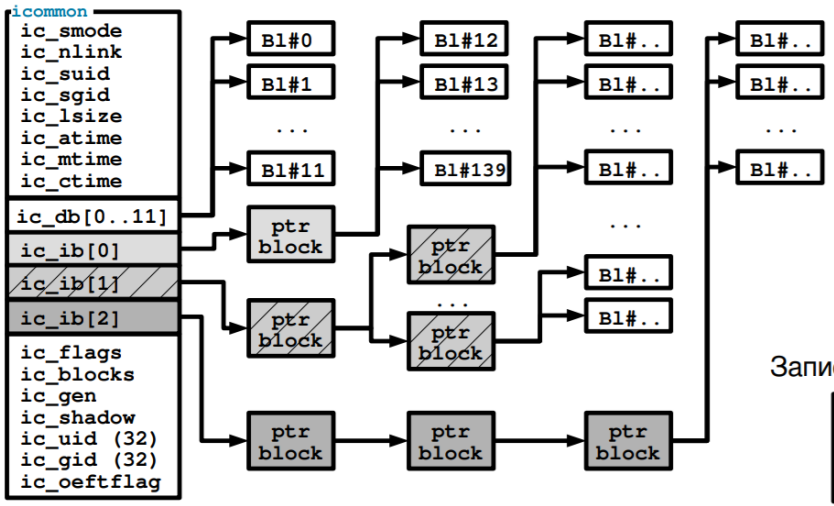
64. Управление файлами в UNIX SVR4

Структура файловой системы UNIX SVR4 UFS

- Файловая система содержит:
  - Загрузчик ОС
  - Суперблок — геометрия и служебные параметры файловой системы
  - Цилиндровые группы — информация о свободых блоках, массив записей о файлах (inode), блоки данных — равномерно распределены по всему дисковому пространству

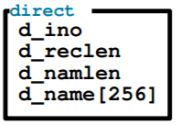


Файловая система UNIX SRV4. Индексированные записи файлов (inode)



- Типы файлов:
  - Обычные файлы
  - Директории
  - Блочные устройства
  - Символьные устройства
  - Именованные каналы (pipes)
  - Жесткие ссылки
  - Символические ссылки

Запись в директории



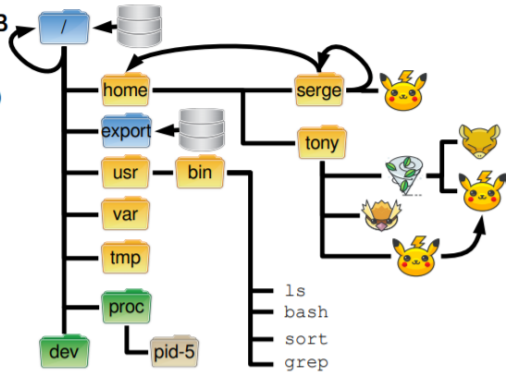
65. Каталоги файлов. Элементы каталога, операции ОС. Look

Каталоги файлов. Элементы каталога.

- Основные:
  - Имя файла, Тип файла (бинарный, текстовый, ...)
  - Организация файла — организация внутренней структуры
- Адресная информация
  - Том (носитель) — указатель на физическое устройство
  - Начальный адрес — номер первого блока файла
  - Занимаемый размер и зарезервированный размер— количество байт в файле и максимальное количество байт
- Информация об управлении доступом
  - Владелец — пользователь, который может управлять файлом
  - Разрешенные действия (чтение, запись, исполнение, поиск, и т.п.)
- Информация об использовании
  - Создатель и дата создания — создатель не обязательно может быть владельцем
  - Дата последнего чтения и последний пользователь, прочитавший файл
  - Дата последнего изменения и последний пользователь, изменивший файл
  - Дата последней резервной копии на другом устройстве
  - Текущее использование — информация о текущих действиях с файлом

Каталог, операции ОС

- Часть элементов каталога может находится в записи о файле, уменьшая его размер
- Каталог может быть полностью или частично загружен в основную память
- Простейшая структура — список записей фиксированной длинны
- Основные операции:
  - Поиск файловых записей,
  - Создание и удаление записей о файле
  - Получение списка записей
  - Обновление каталога, изменение записи
- Обычно структура каталогов — n-уровневая иерархия, дерево, связанный граф



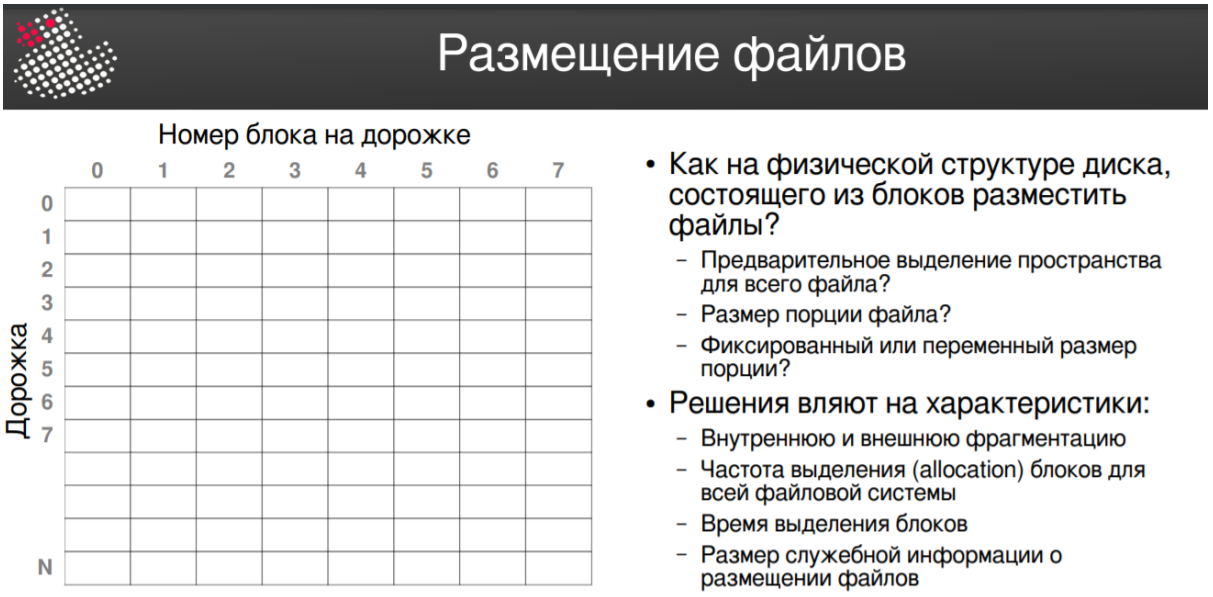
- Именование:
  - Имя файла
  - Имя, включающее полный путь от корня
  - Имя, относительно текущего каталога

Существует 2 типа фрагментации: Внутренняя: когда выделяется больше памяти, чем запрашивалось, избыток памяти не используется; Внешняя: свободная память в процессе выделения или освобождения разделяется на мелкие блоки и в результате не обслуживаются некоторые запросы на выделение памяти.

66. Размещение записей и файлов в блоках данных. Сложность и типы организации размещения.

- Запись — логическая единица доступа к структурированному файлу
  - ОС осуществляет ввод-вывод блоками
- Какой должен быть размер блок относительно размера записи?
  - Блок больше → Больше записей передано за одну операцию ввода-вывода
  - При случайном доступе — передаем записи, которые не используются
  - Большим блокам нужны большие буферы ОС

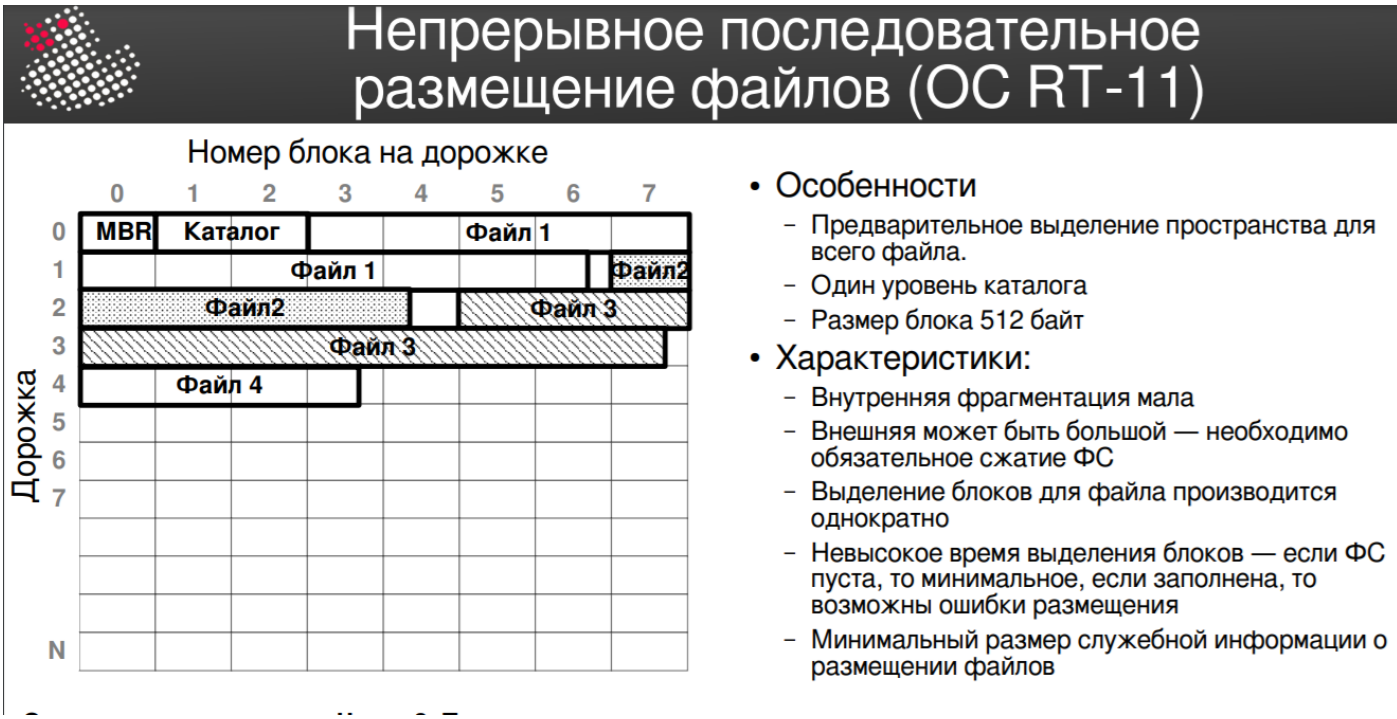
Чем больше блок - тем больше объем данных, переданный за одну операцию ввода-вывода => тем меньше нагрузка на ОС
Обычно размер блока кратен размеру страницы



- Как на физической структуре диска, состоящего из блоков разместить файлы?
  - Предварительное выделение пространства для всего файла?
  - Размер порции файла?
  - Фиксированный или переменный размер порции?
- Решения вляют на характеристики:
  - Внутреннюю и внешнюю фрагментацию
  - Частота выделения (allocation) блоков для всей файловой системы
  - Время выделения блоков
  - Размер служебной информации о размещении файлов

Больше про размещение в следующих билетах.

67. Непрерывное размещение файлов (на примере ОС RT-11)

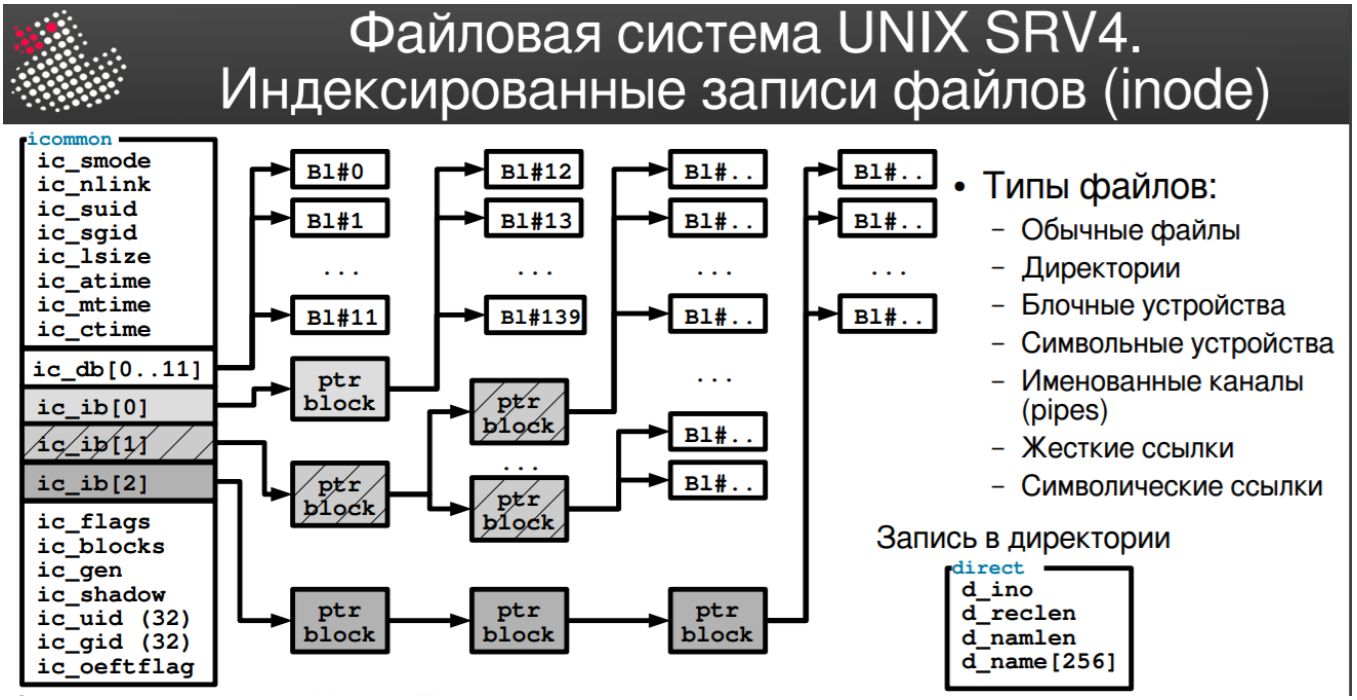


Плюс - быстрое последовательное чтение файлов
Минус - дыры после удаления файлов

68. Цепочечное размещение файлов (на примере DOS FAT)

Файл разбивается на кластеры одинакового размера, которые затем размещаются в памяти.
Таблица указывает на следующий кластер для каждого кластера.
Удобно заменять файлы, поскольку можно заменять кластеры последовательно.

69. Индексированное размещение (на примере файловой системы UNIX UFS) .



ic_db - где смотреть запись
ic_id[n] - указатели на блоки с разной косвенностью, итоговый пункт назначения-блок с данными.
Индексированная запись - потому что указывает индексом на блок, который нужно использовать.

Дополнительные вопросы для кафедры ВТ.

70. Linux: стандартные средства для наблюдения счетчиков ядра.

Стандартные средства для наблюдения счетчиков ядра

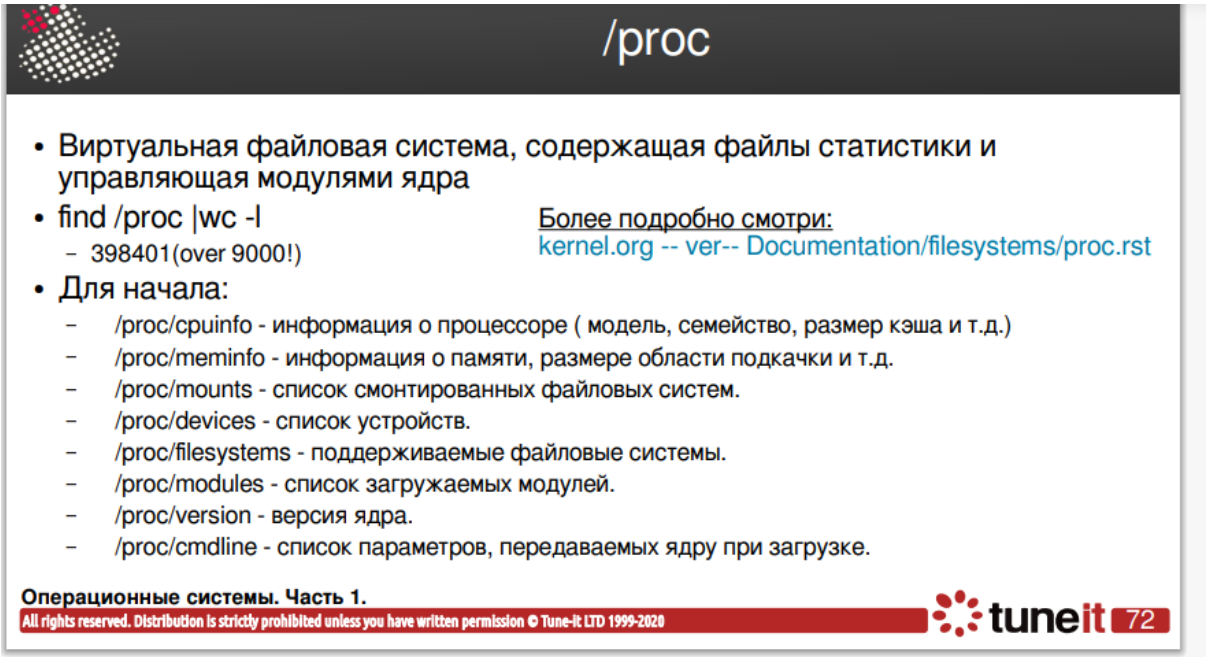
- sar (system activity reporter): общесистемное средство, собирающая статистику по пейджингу (-B) и свопингу (-W), вводу-выводу (-b,-d), смонтированным системам (-F), прерываниям (-I), управлению питанием (-m), сети (-n), процессорам (-P,-u), очереди процессов и загрузке (-q,-w), памяти (-r), области подкачки (-s), терминалам (-y)
- Можно настроить сбор исторических результатов (crontab)
- Пример: sar -q 1 1 (*одно измерение за одну секунду*)

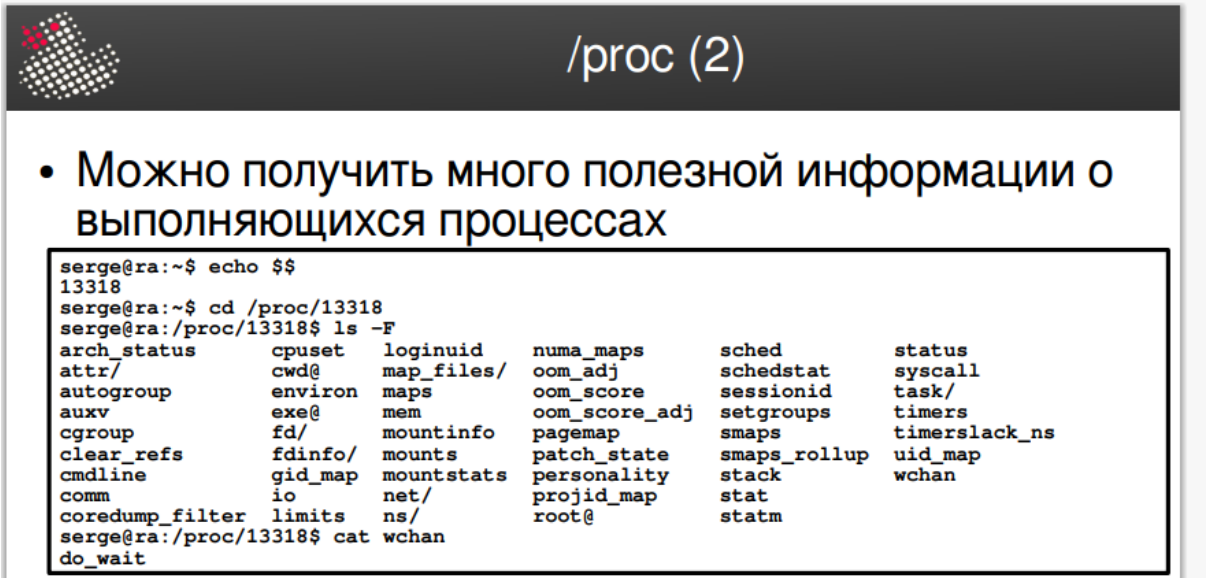
```
Linux 5.4.0-47-generic (ra)   01.10.2020   _x86_64_ (4 CPU)

14:35:36      runq-sz  plist-sz   ldavg-1   ldavg-5  ldavg-15   blocked
14:35:37            0      1034      1,44      1,70      1,75         0
Average:            0      1034      1,44      1,70      1,75         0
```


Стандартные средства для наблюдения счетчиков ядра (2)

- Процессор: ps, top, tiptop, turbostat, rdmsr, numastat, uptime
- Виртуальная память: vmstat, slabtop, pidstat, free
- Дисковая подсистема: iostat, iotop, blktrace
- Сеть: netstat, tcpdump, iptraf, ethtool, nicstat, ip
- Интерактивные (типа top) или с указанием количества запуска и интервала (типа sar)
- Некоторые работают только с правами root!

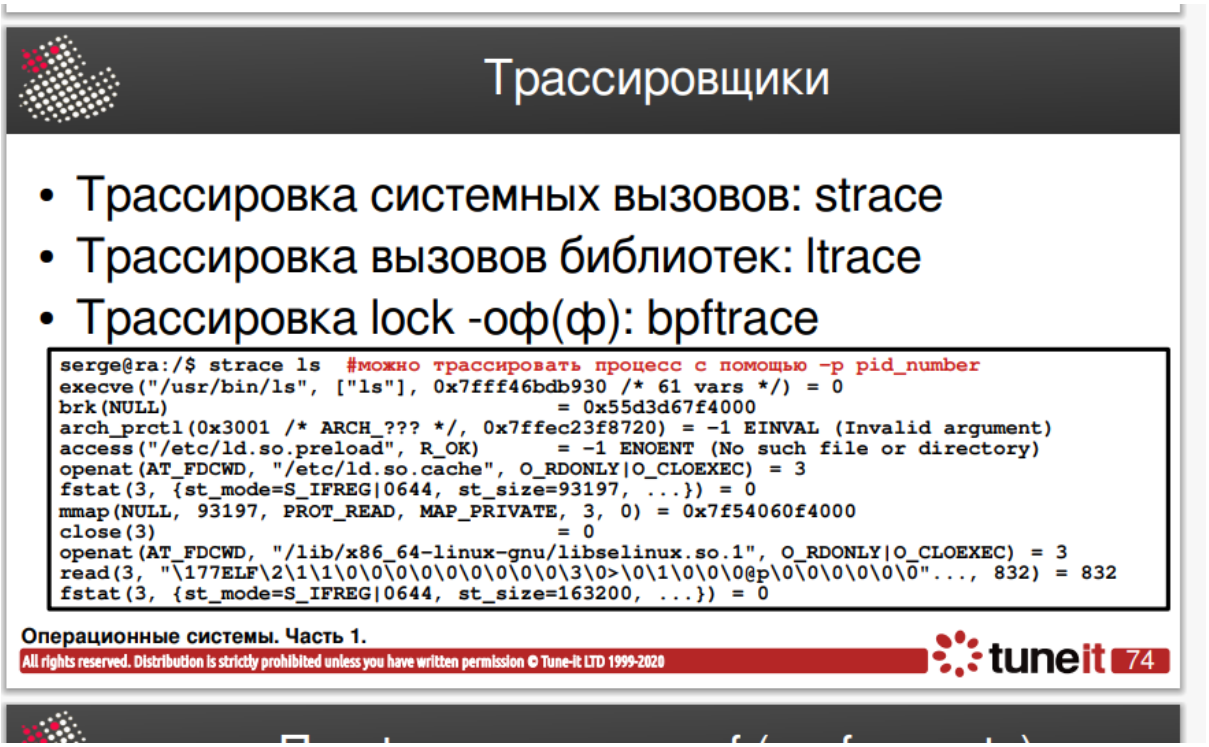
71. Linux: файловая система /proc.



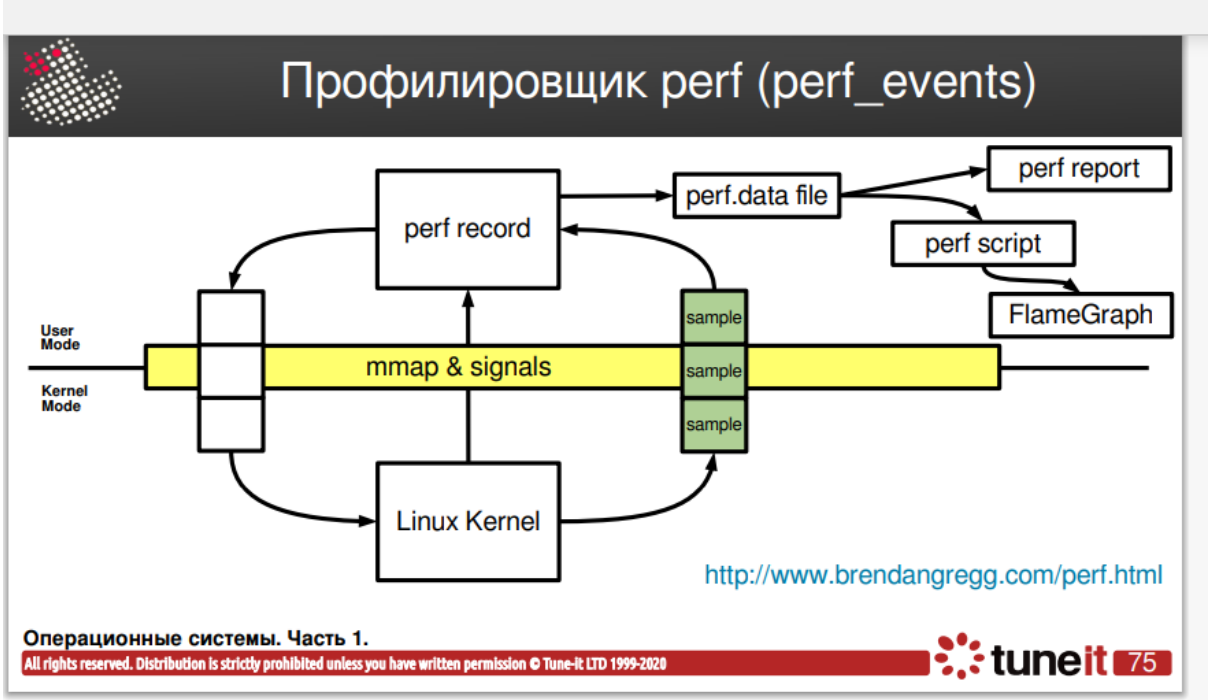


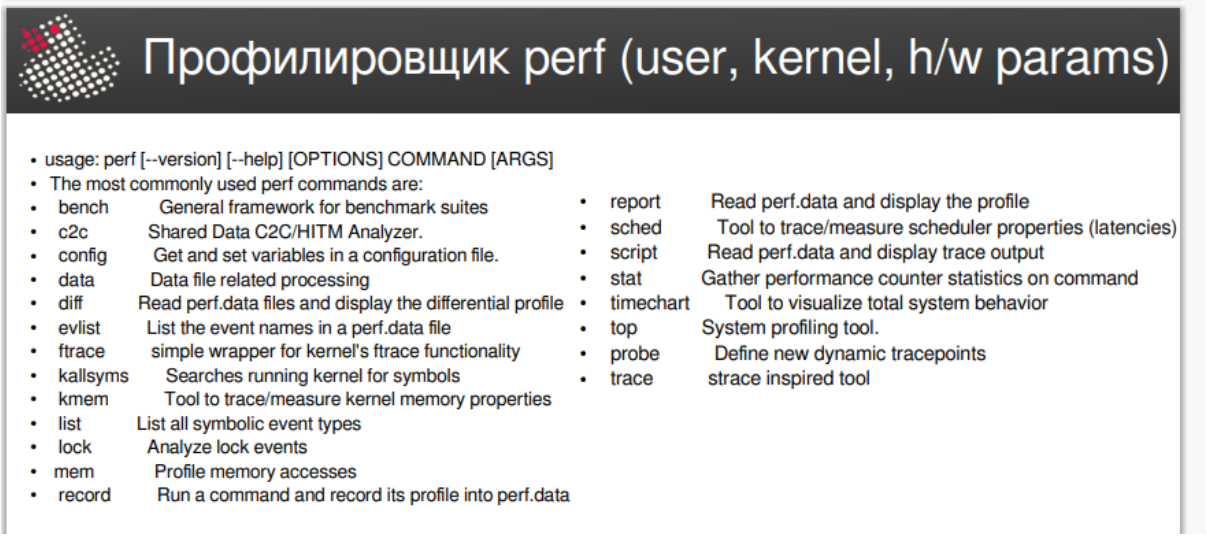
Documentation/filesystems/proc.rst

72. Linux: трассировщики системных вызовов и библиотек.



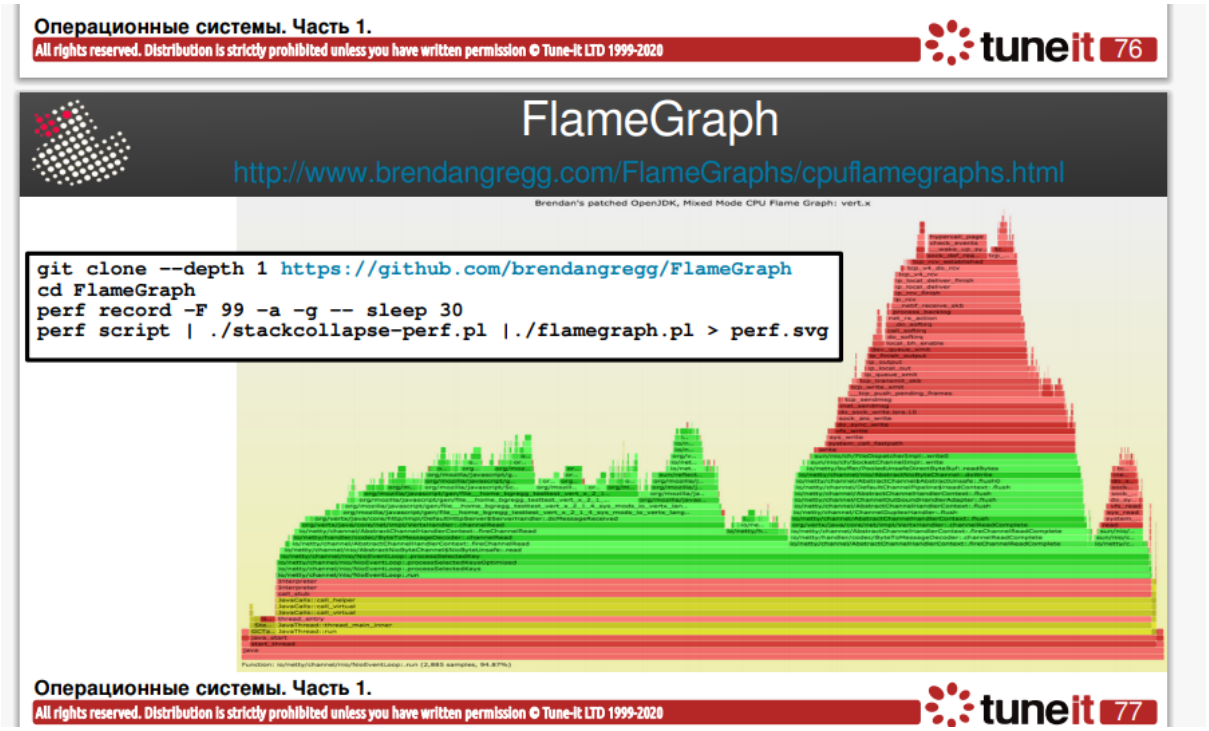
73. Linux: Профилировщик perf и FlameGraph.





http://www.brendangregg.com/perf.html

https://star-wiki.ru/wiki/Perf_(Linux)



http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

74. Linux: SystemTap.

https://m.habr.com/ru/post/77502/

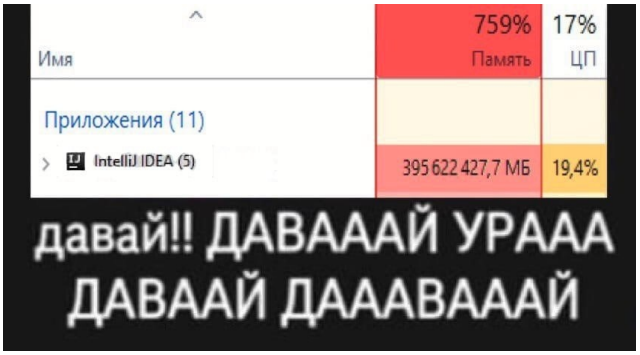
75. Linux: Отладчик ядра.

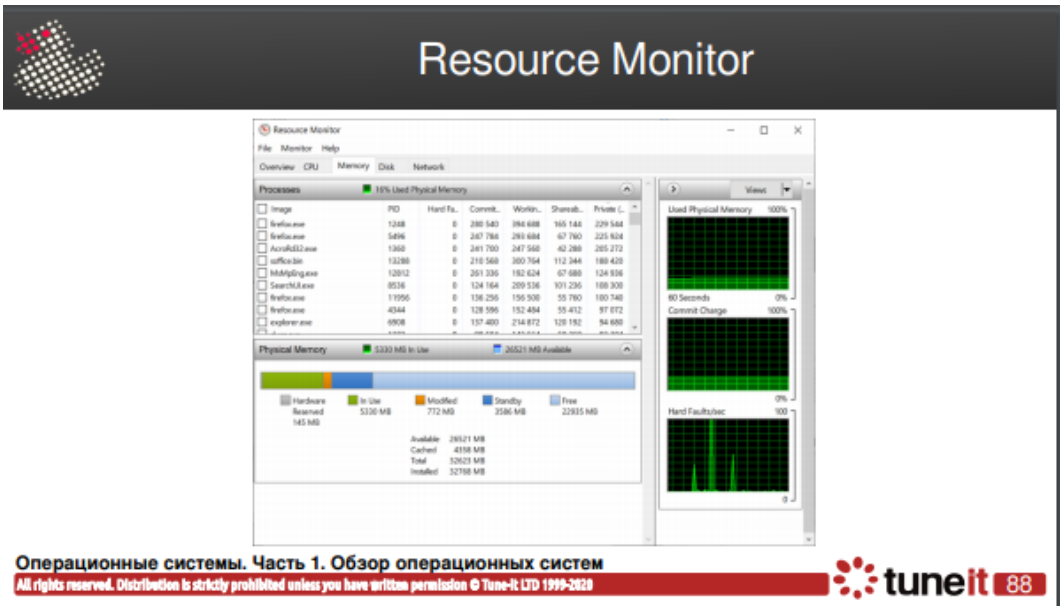
1) Лучше этим не пользоваться чтобы не поломать свою ос, как говорил Миша Филиппов не играйте с ядром а то можно и проиграть
2) Есть команды ГО (запуск) и ? (хелп)
3) Это типа как гдб но для системных вызовов
4) Можно запускать удаленно
5) Это один из самых конченных билетов так что если он вам выпал вы плохо молились своему богу

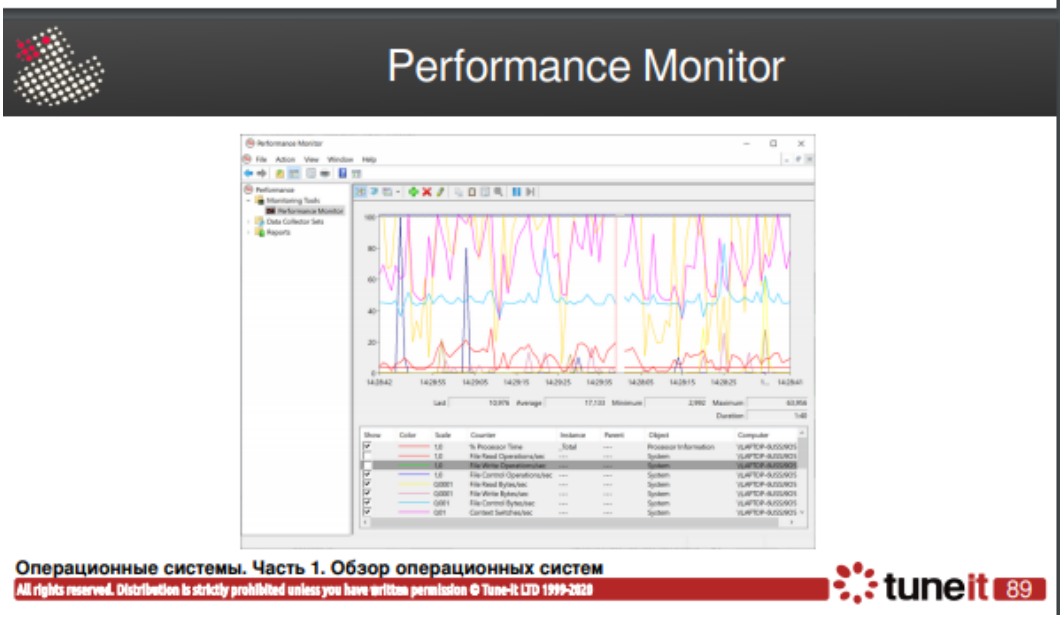
   (вот спасибо!)

https://russianblogs.com/article/19411334724/

.76. Windows: стандартные отладочные средства.

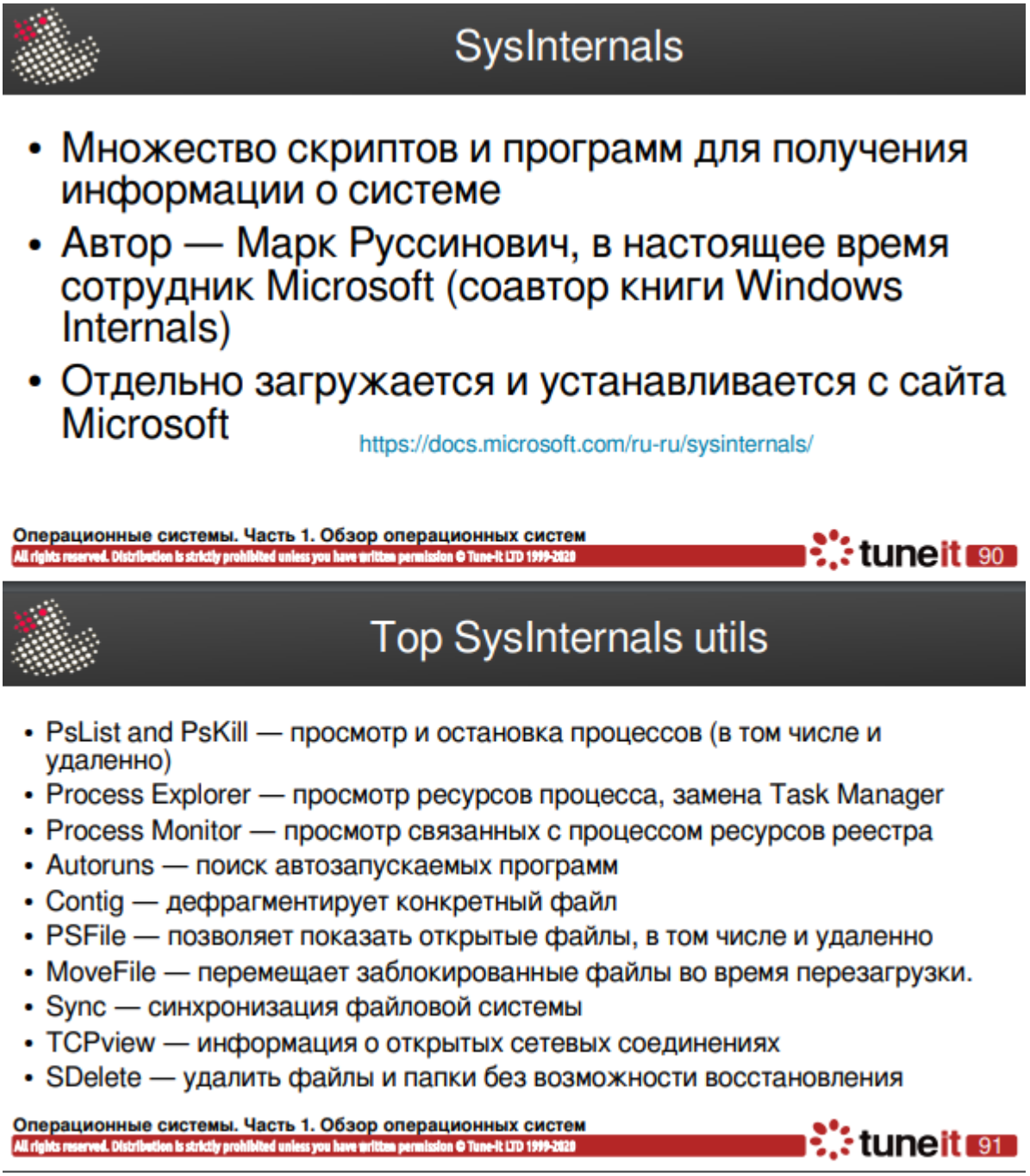






https://anydifferencebetween.com/performance-monitor-vs-resource-monitor/

77. Windows: утилиты SysInternals



https://ru.wikipedia.org/wiki/Sysinternals

78. Windows: отладчики WinDbg и KD

Абсолютно то же самое, что и отладчик ядра в линухе

.79. Аппаратная поддержка взаимных исключений.

1) Отключение прерываний

2) Атомарные операции на уровне ядра
(Столлингс стр. 285)

(задача о парикмахерской - проблема с очередью обслуживания)

.80. Эволюция подхода к блокировке (Столлингс, гл. 5.1)

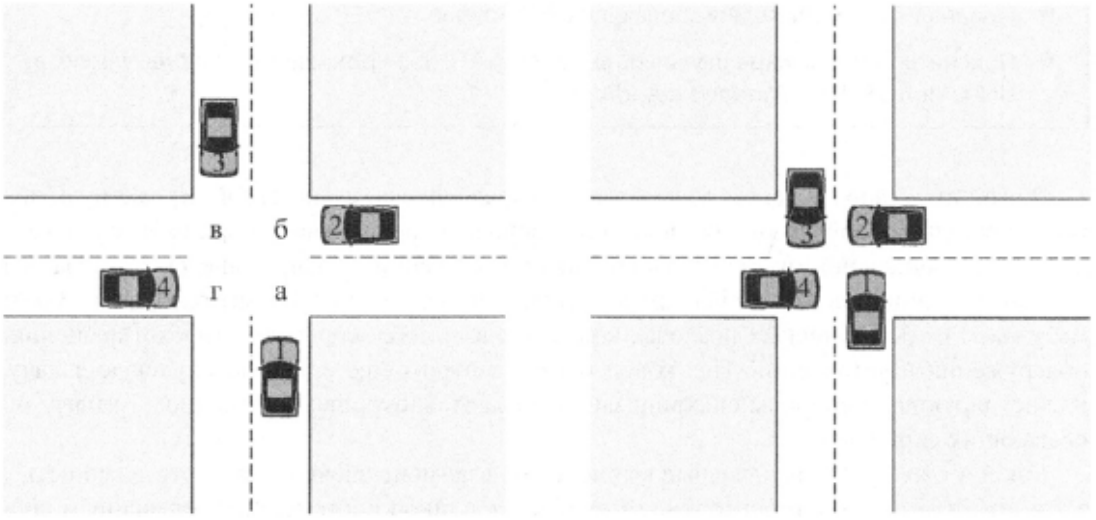
?

81. Принципы взаимного блокирования (Столлингс, гл. 6.1)

История про машинки

Все взаимоблокировки предполагают наличие конфликта в борьбе за ресурсы между двумя или несколькими процессами. Наиболее ярким примером может служить транспортная взаимоблокировка. На рис. 6.1 показана ситуация, когда четыре автомобиля должны примерно одновременно пересечь перекресток. Четыре квадранта перекрестка представляют собой ресурсы, которые требуются процессам. В частности, для успешного пересечения перекрестка всеми четырьмя автомобилями необходимые ресурсы выглядят следующим образом.

- Автомобилю 1, движущемуся на север, нужны квадранты *а* и *б*.
- Автомобилю 2, движущемуся на запад, нужны квадранты *б* и *в*.
- Автомобилю 3, движущемуся на юг, нужны квадранты *в* и *г*.
- Автомобилю 4, движущемуся на восток, нужны квадранты *г* и *а*.



а) Взаимоблокировка возможна б) Взаимоблокировка произошла

Рис. 6.1. Пример взаимоблокировки

A

82. Предотвращения взаимоблокировок, устранение взаимоблокировок, обнаружение блокировок. (Столлингс, гл. 6.2, 6.3, 6.4)

Для того чтобы взаимоблокировка стала возможной, требуются три условия.

1. **Взаимные исключения**. Одновременно использовать ресурс может только один процесс.
2. **Удержание и ожидание**. Процесс может удерживать выделенные ресурсы во время ожидания других ресурсов.
3. **Отсутствие перераспределения**. Ресурс не может быть принудительно отобран у удерживающего его процесса.

Кроме перечисленных трех условий, необходимых, но не достаточных, для реального осуществления взаимоблокировки, требуется выполнение четвертого условия.

4. **Циклическое ожидание**. Существует замкнутая цепь процессов, каждый из которых удерживает как минимум один ресурс, необходимый процессу, следующему в цепи после данного (см. рис. 6.5, *в* и 6.6).

83. Задача об обедающих философах (Столлингс, гл. 6.6)

Теперь рассмотрим задачу об обедающих философах, представленную Дейкстрой в [65]. Итак, в некотором царстве, в некотором государстве жили вместе пять философов. Жизнь каждого из них проходила в основном в размышлениях, прерываемых приемом пищи. Философы давно сошлись во мнении, что только спагетти в состоянии восстанавливать их подточенные непрерывными размышлениями силы.

Питались они за одним круглым столом (рис. 6.11), на который помещались большое блюдо со спагетти, пять тарелок, по одной для каждого философа, и пять вилок. Проголодавшийся философ садится на свое место за столом и, пользуясь двумя вилками, приступает к еде. Задача состоит в том, чтобы разработать ритуал (читай — алгоритм) обеда, который обеспечивает взаимоисключения (два философа не могут одновременно пользоваться одной вилкой) и не допускает взаимоблокировок и голодания (обратите внимание, насколько уместным оказался этот термин в данной задаче!).

Эта задача Дейкстры может показаться не очень важной, но она очень хорошо иллюстрирует проблемы взаимоблокировок и голодания. Кроме того, при решении данной задачи приходится сталкиваться со многими трудностями в организации параллельных вычислений (см., например, [89]). Задача об обедающих философах может рассматриваться как типичная задача, возникающая в многопоточных приложениях при работе с совместно используемыми ресурсами и, соответственно, может выступать в качестве тестовой при разработке новых подходов к проблеме синхронизации.

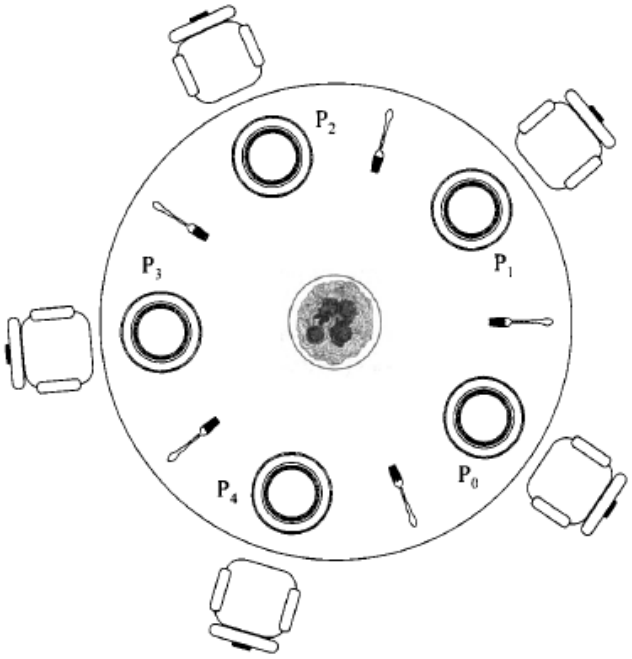
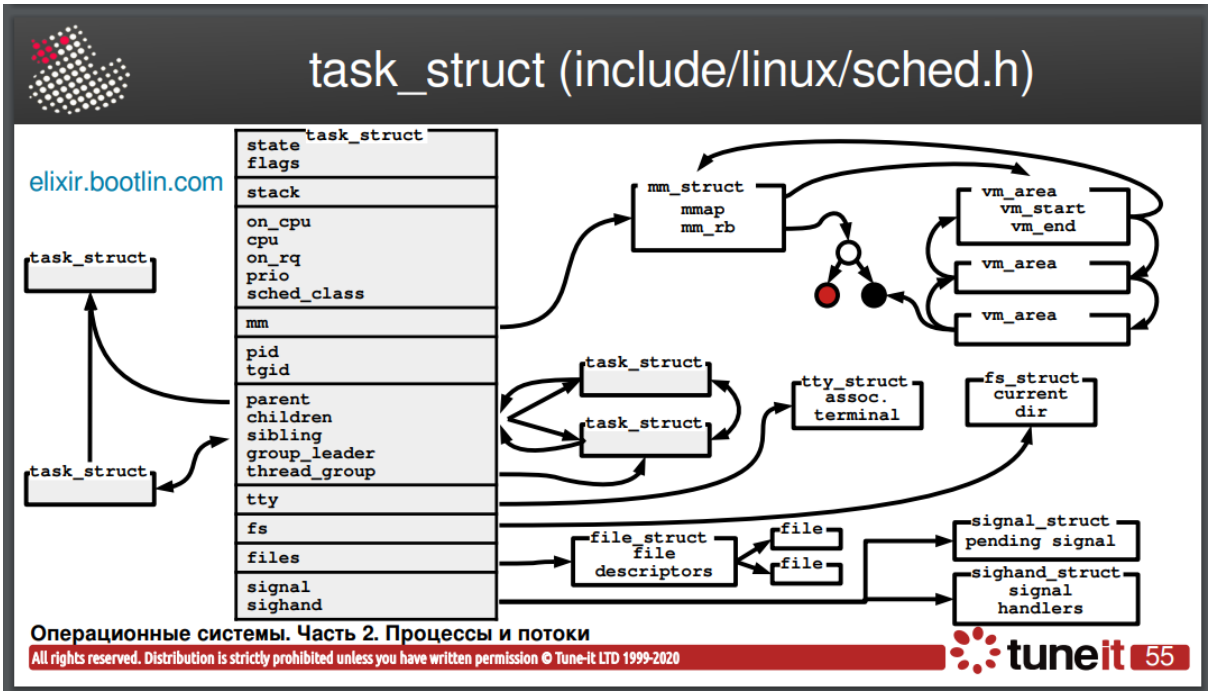


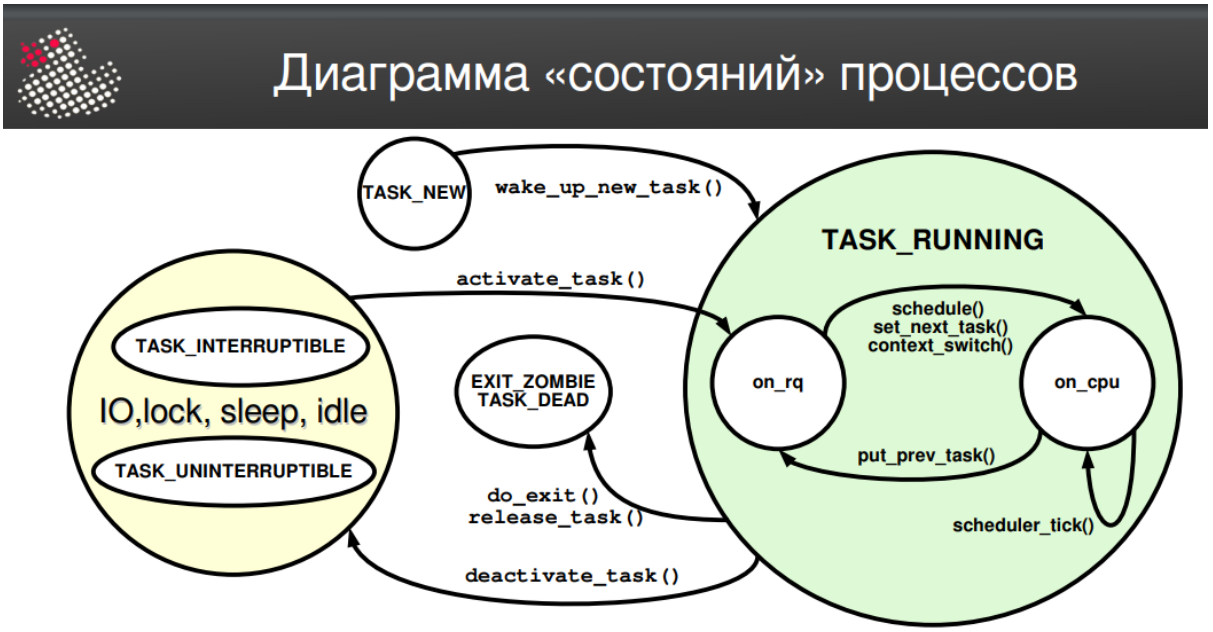
Рис. 6.11. Обеденный стол философов

84. Процессы в Linux: структура task_struct, поля струткуры, связь с другими структурами ядра.

В ядре Linux процесс представлен довольно большой структурой `task_struct` (дескриптором процесса). Помимо самой необходимой для описания процесса информации эта структура содержит массу других данных, используемых для учета и связи с другими процессами (родительскими и порожденными). Полное описание структуры `task_struct` выходит за рамки статьи, однако, ее фрагмент, содержащий упомянутые в статье элементы, приведен в листинге 1. Стоит заметить, что структура `task_struct` объявлена в файле ./linux/include/linux/sched.h.

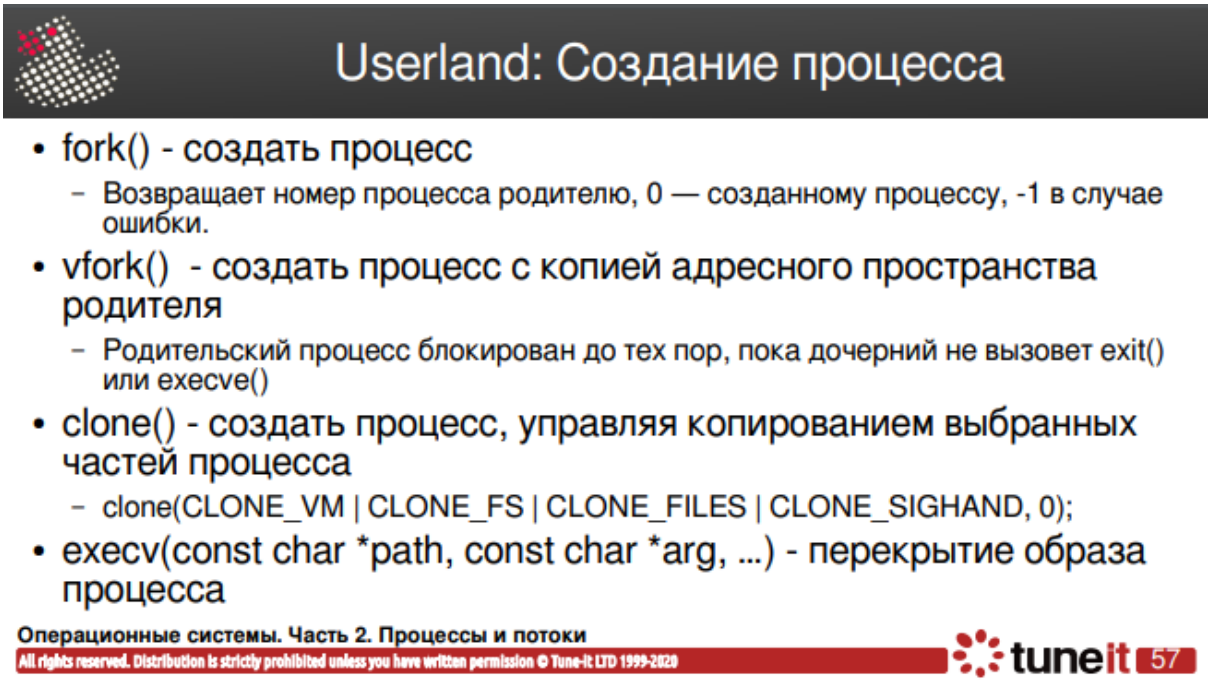


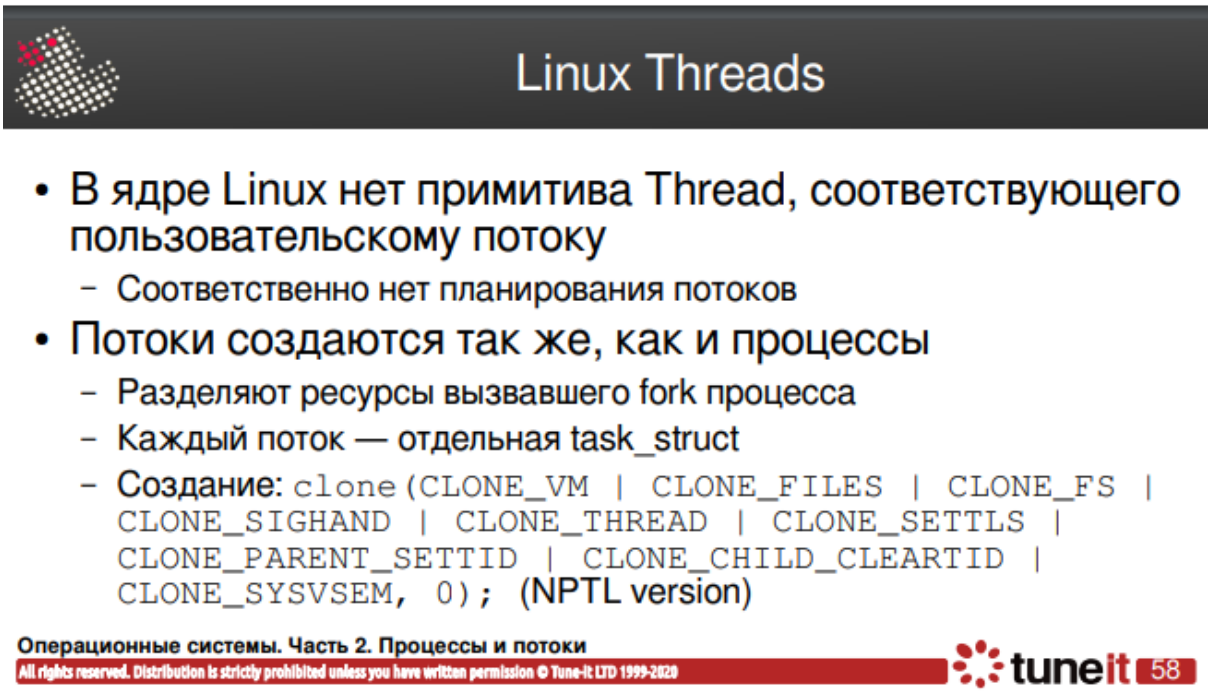
85. Диаграмма состояния процесса Linux.



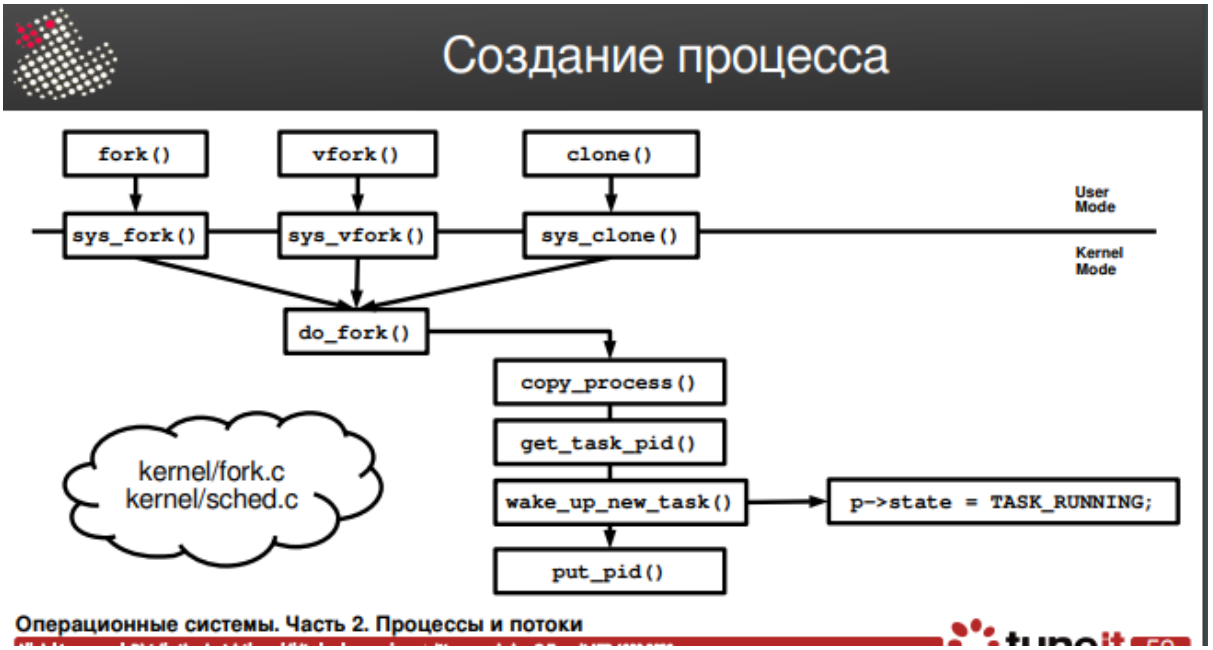
https://ravesli.com/processes-v-linux/

86. Создание процесса Linux на уровне пользовательского процесса.

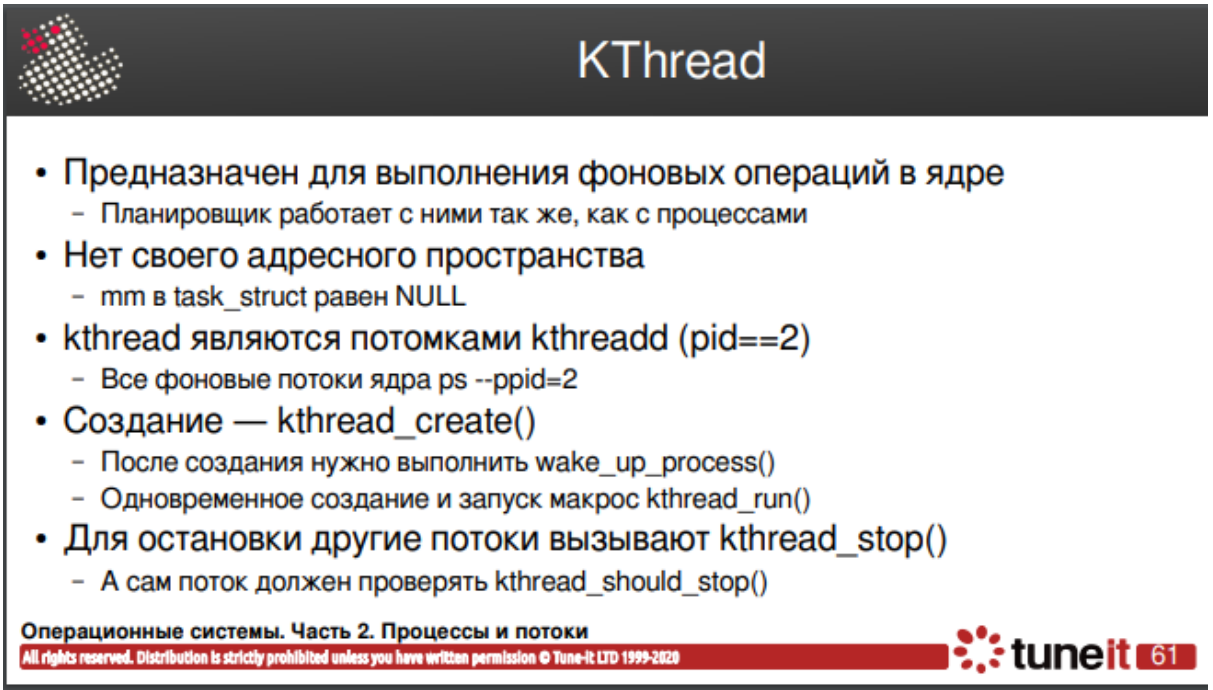


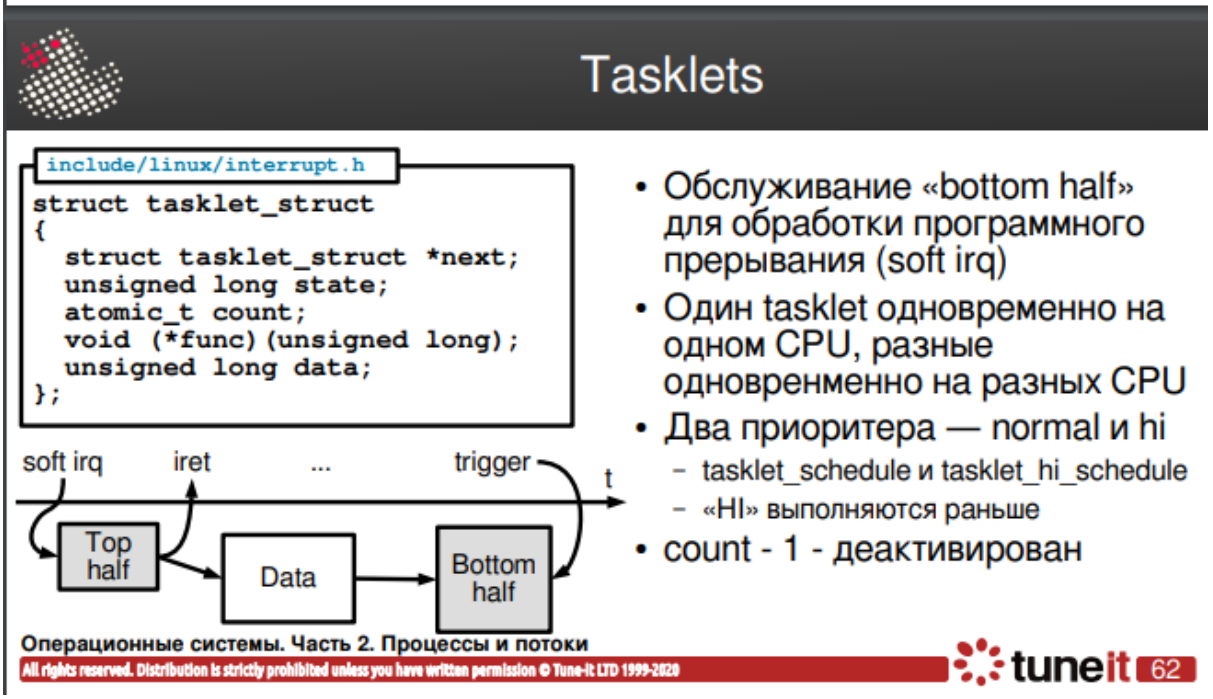


87. Создание и завершение процесса Linux на уровне ядра. Вызываемые функции.



88. Особенности реализации потоков в Linux. KThread. Tasklet.





https://habr.com/ru/company/embox/blog/244071/

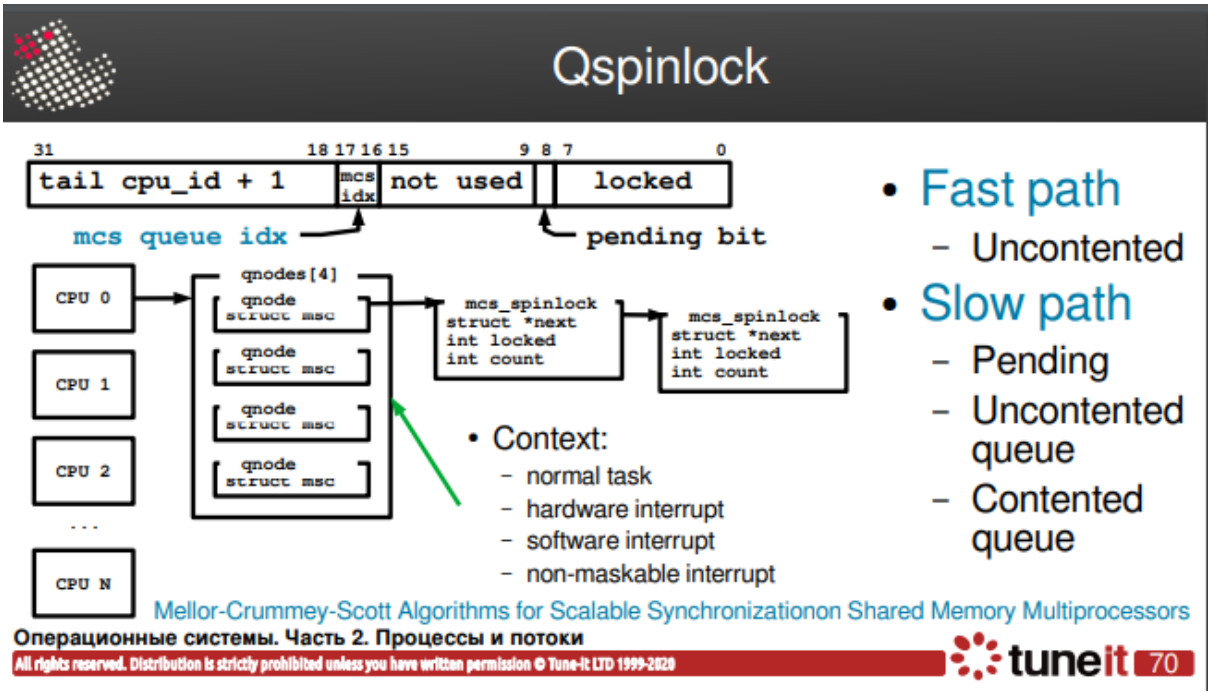
01

,89. Примитивы синхронизации Linux. Spinlock и qspinlock.

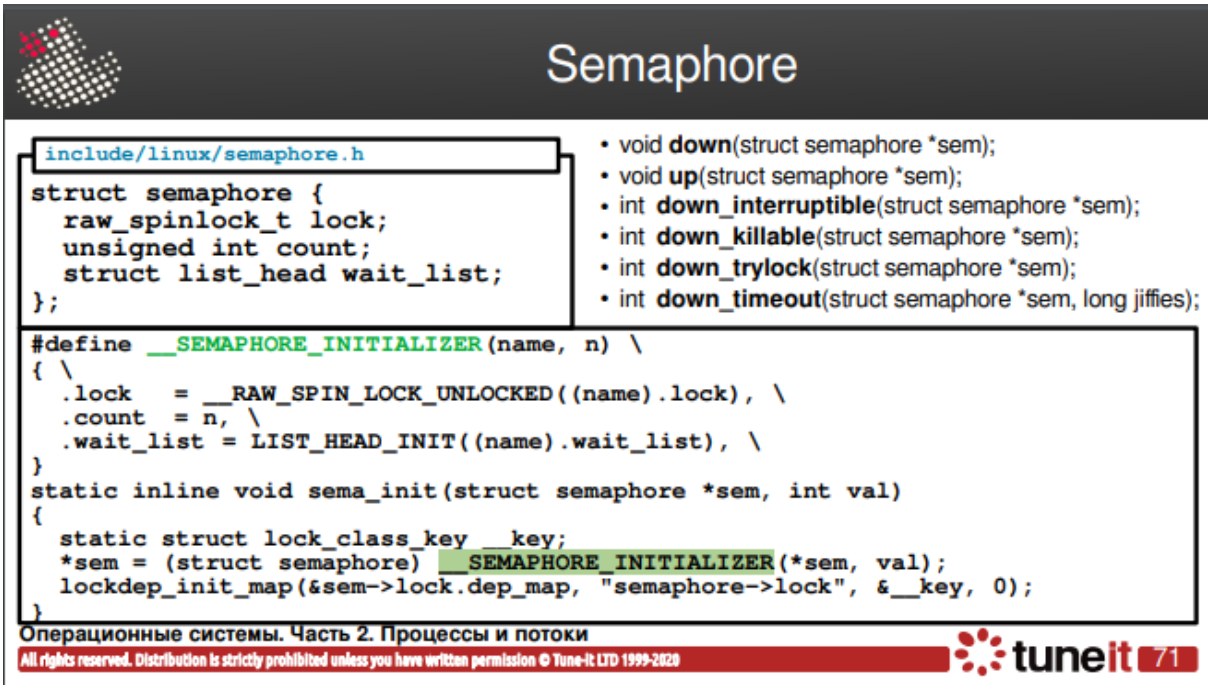
Спин-блокировка или **спинлок** (англ. *spinlock* — циклическая блокировка) — низкоуровневый примитив синхронизации[1], применяемый в многопроцессорных системах для реализации взаимного исключения исполнения критических участков кода с использованием цикла активного ожидания[2]. Применяется в случаях, когда ожидание захвата блокировки предполагается недолгим[2], либо если контекст выполнения не позволяет переходить в заблокированное состояние[3].

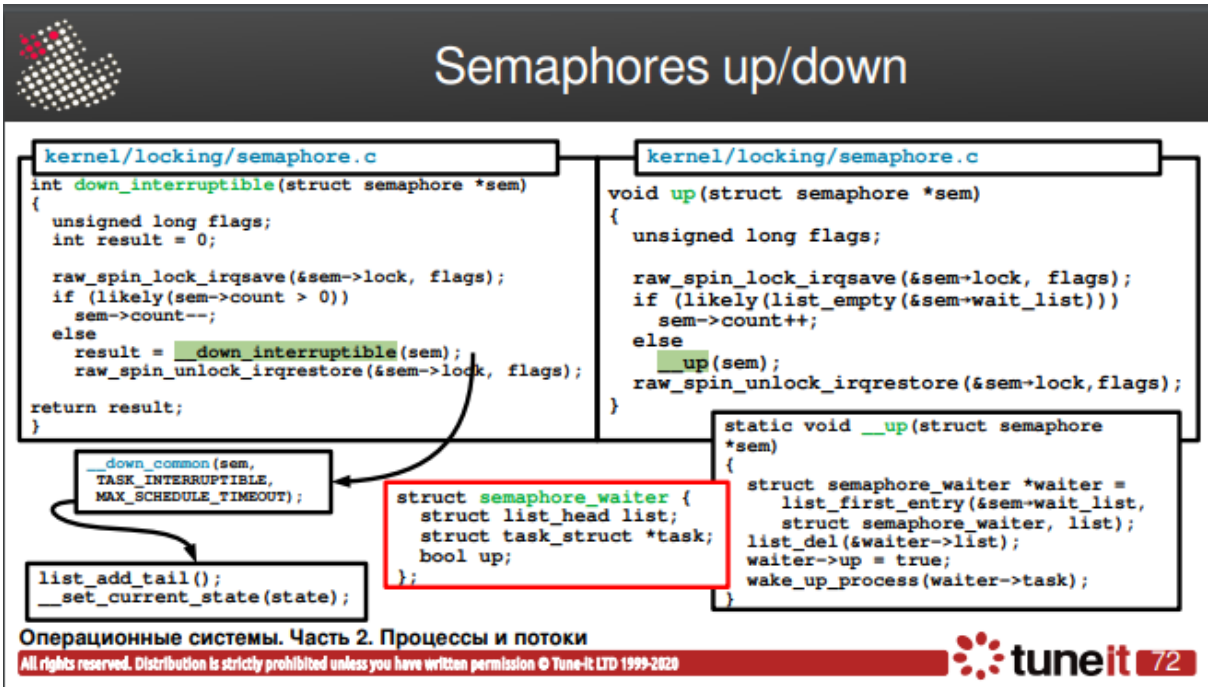
Спин-блокировки являются аналогами мьютексов, позволяющими тратить меньше времени на процедуру блокировки потока, поскольку не требуется переводить поток в заблокированное состояние. В случае мьютексов может потребоваться задействование планировщика с переводом потока в другое состояние и добавлением его в список потоков, ожидающих разблокировки. Спин-блокировки не задействуют планировщик и используют цикл активного ожидания без изменения состояния потока, что приводит к трате процессорного времени на ожидание освобождения блокировки другим потоком. Типовой реализацией спин-блокировки является простая циклическая проверка переменной спин-блокировки на доступность[1]

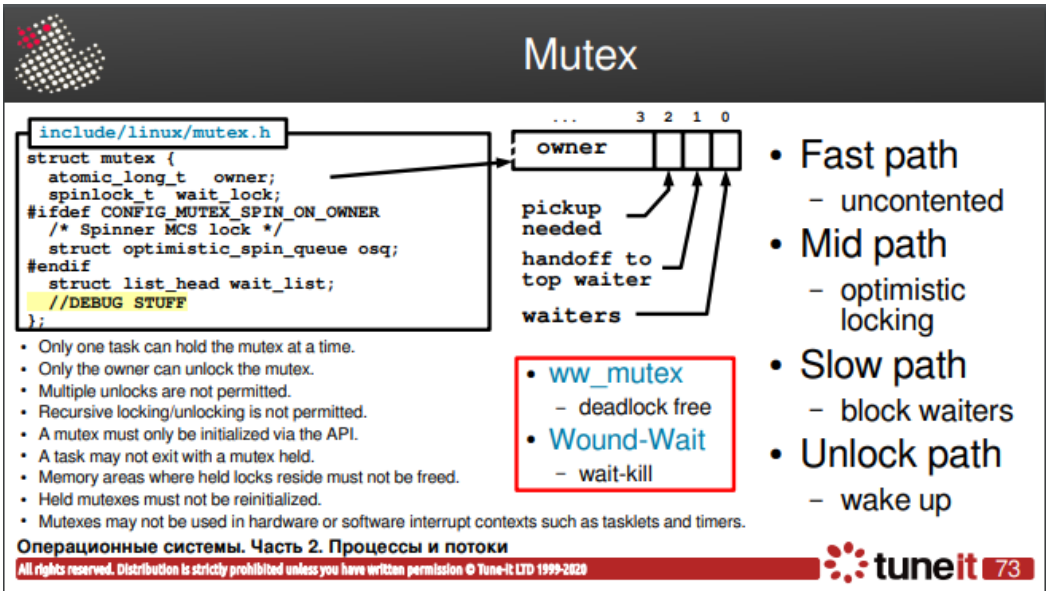
Qspinlock - spinlock с очередью



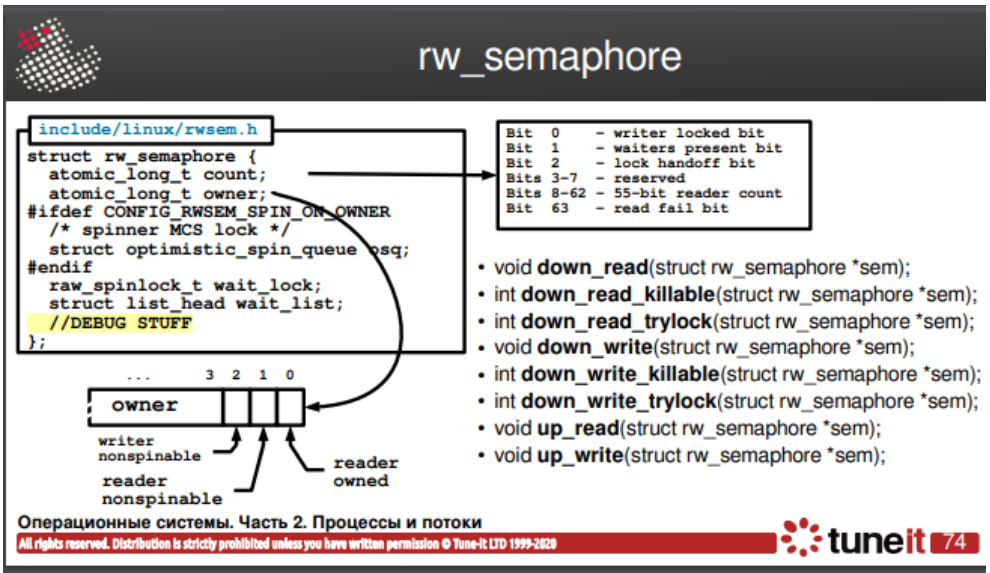
90. Примитивы синхронизации Linux. Semaphore и Mutex.

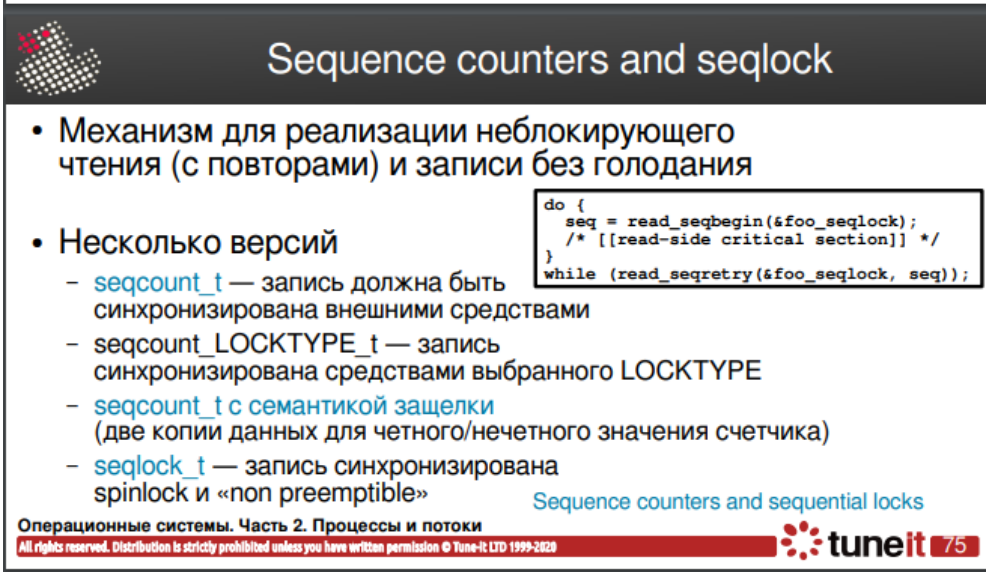




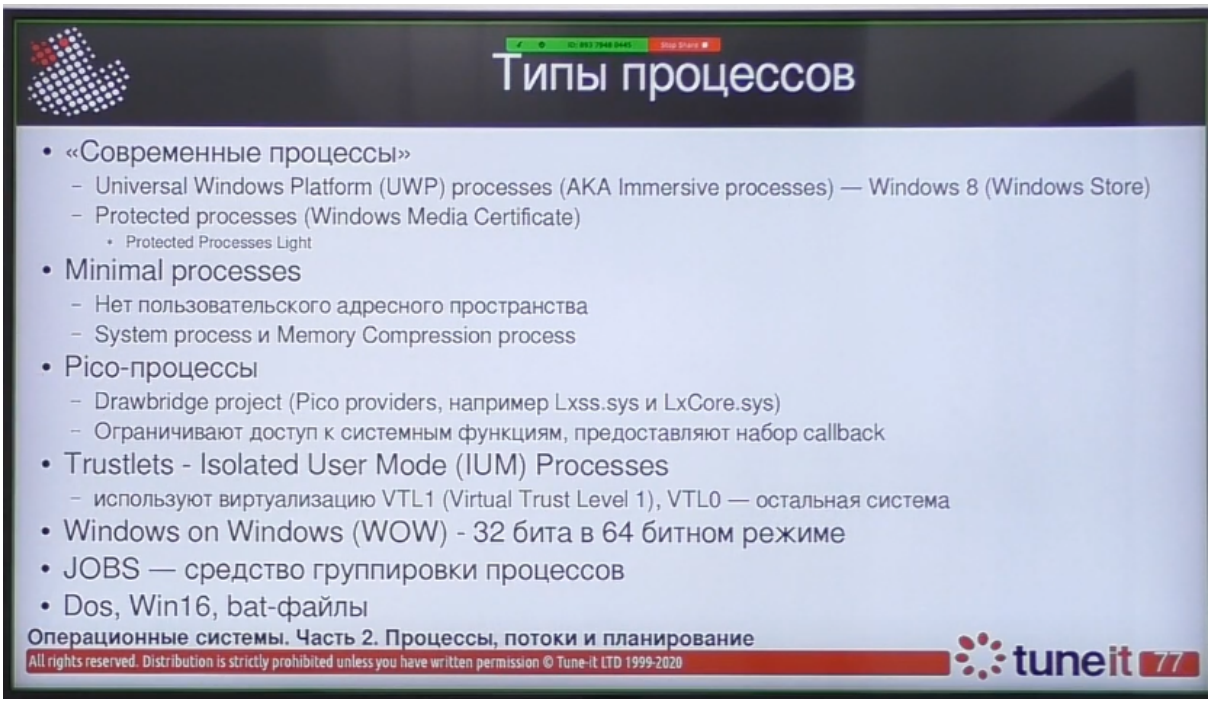


.91. Примитивы синхронизации Linux. rw_semaphore, seqloc

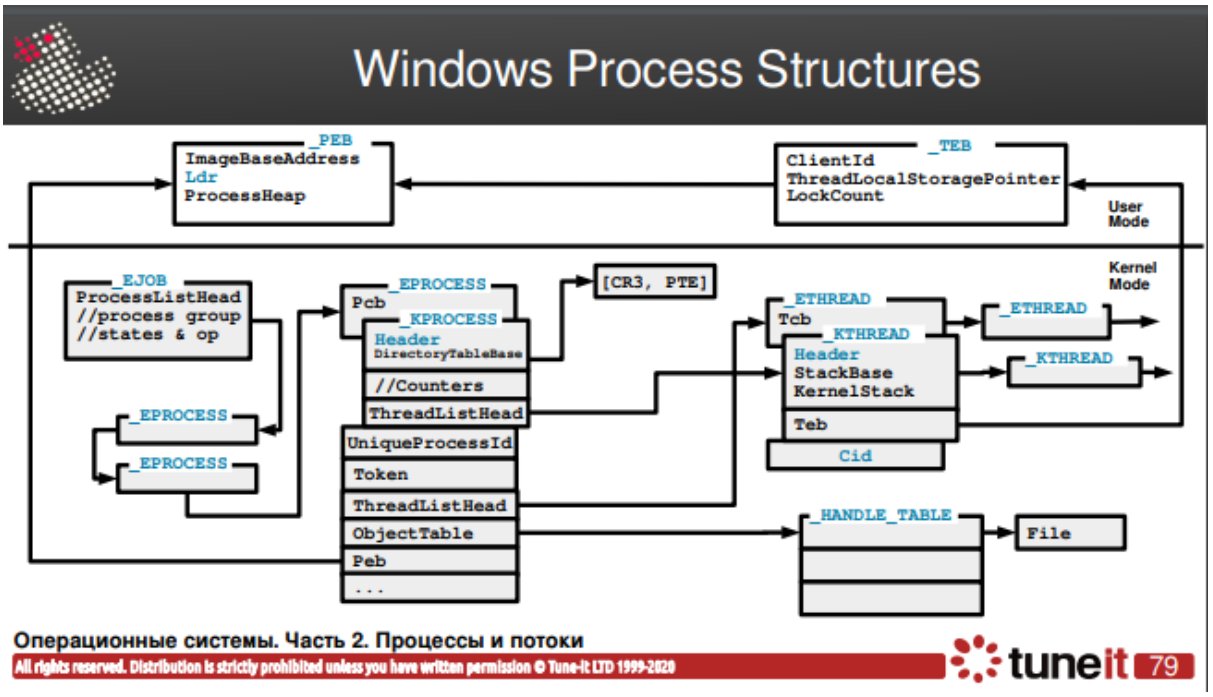




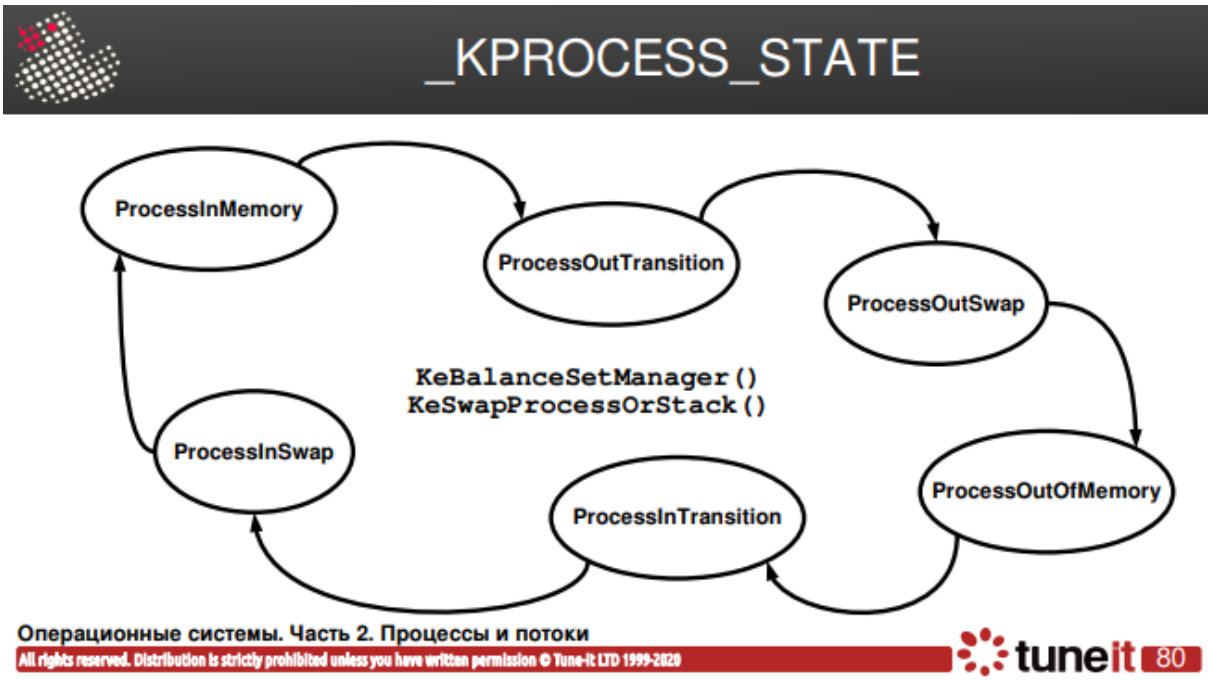
.92. Типы процессов и потоков Windows.

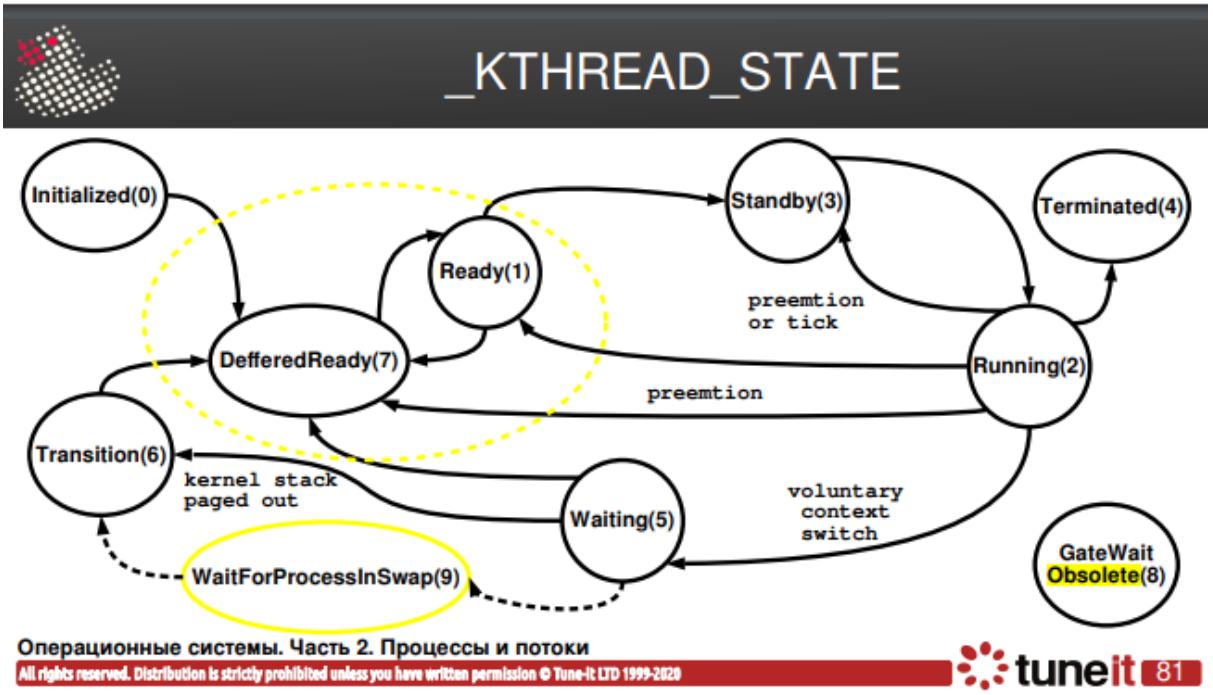


.93. Структура процесса и потока в Windows. Поля структур.



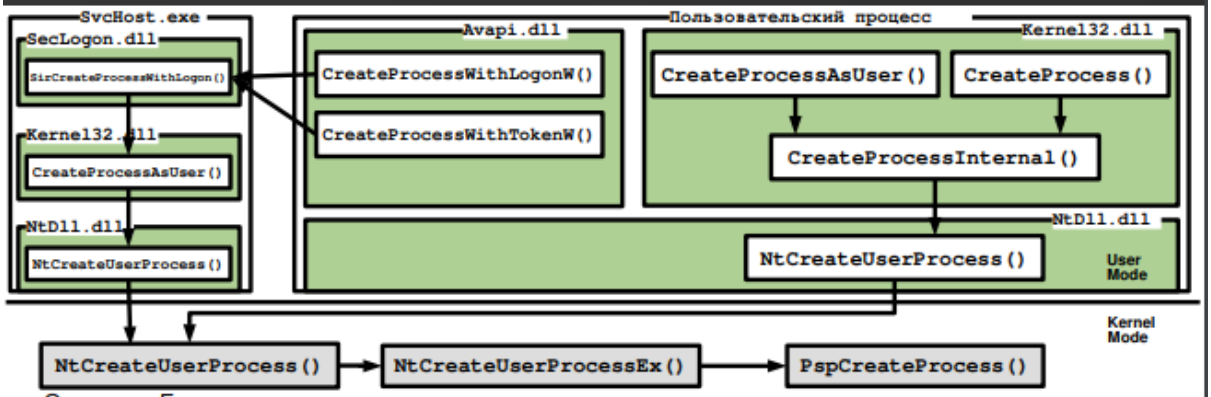
.94. Диаграммы состояний процесса и потока Windows





.95. Создание и завершение процесса Windows.

Создание процесса



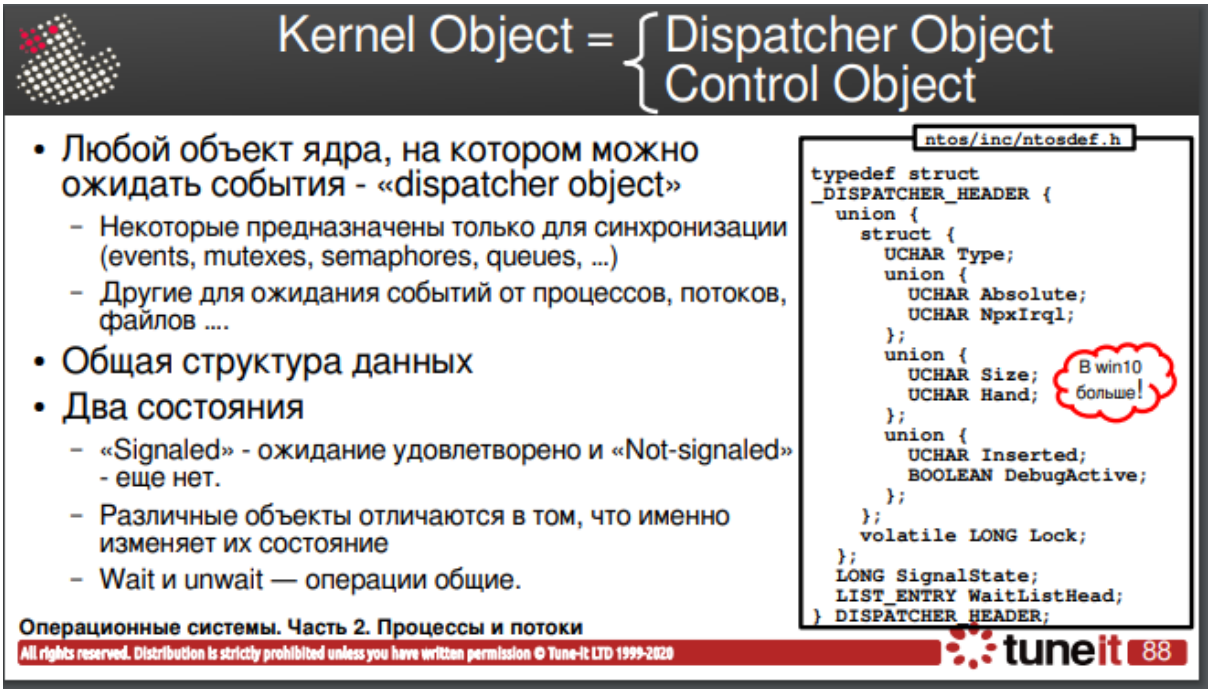
- Открывает Exe
- Создает память
- Создает структуру процесса
- Создает основной поток

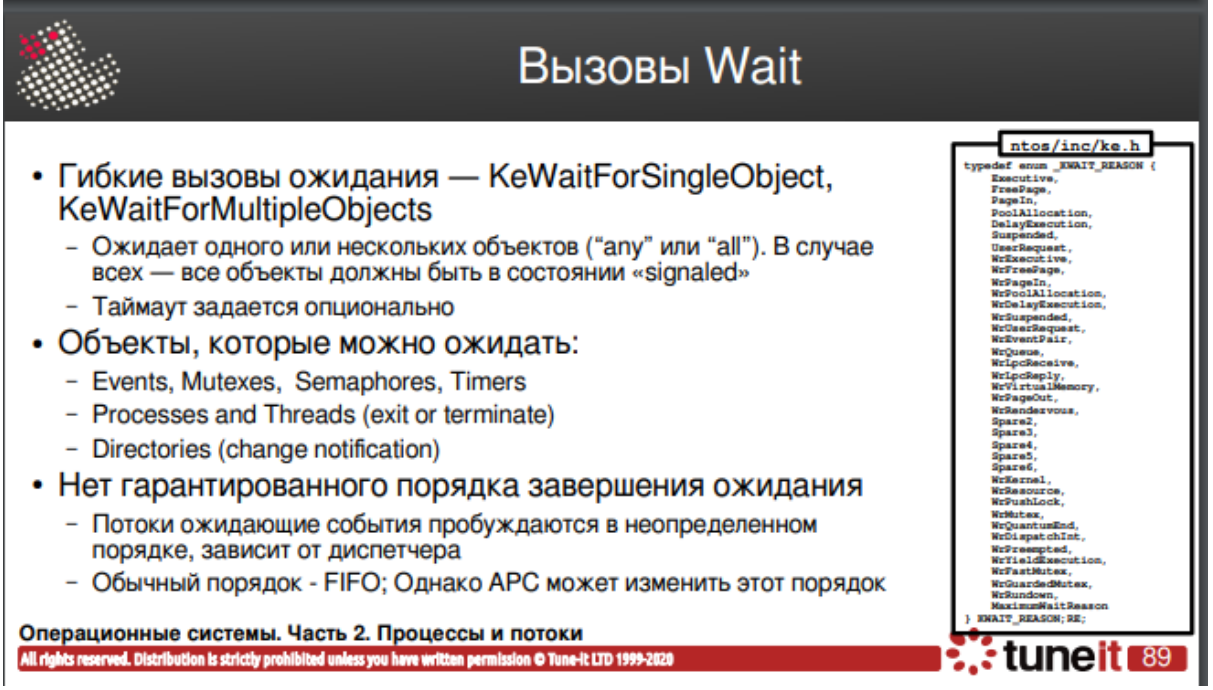
- Уведомляет Windows о создании
- Запускает новый поток и завершается. В новом потоке:
  - Финальные настройки
  - Запуск со стартового адреса в образе процесса

Завершение процессов

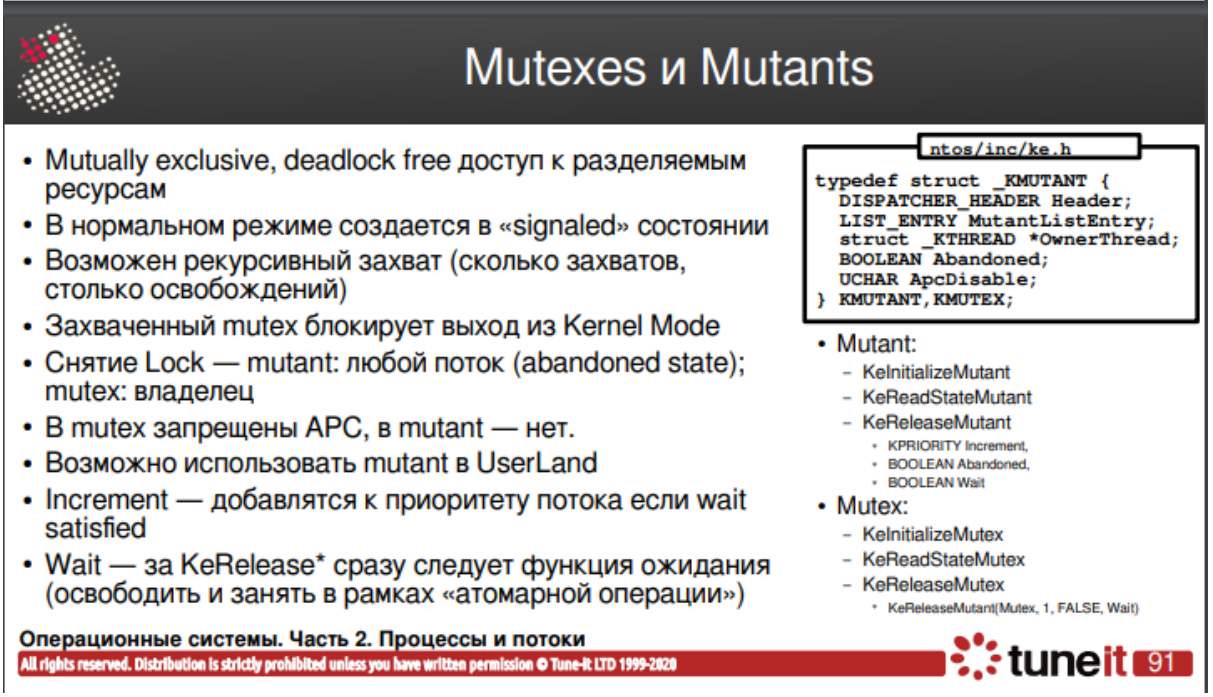
- Корректное завершение - Exit Process()
- Прекращение функционирования процесса другим процессом TerminateProcess()
- Последовательность:
  1. Оповещение DLL
     - Если не использован TerminateProcess()
  2. Закрываются все handles и kernel objects
  3. Закрываются все активные потоки
  4. Код возврата изменяется с STILL_ACTIVE на указанный
  5. Когда все ссылки на процесс == 0, объект процесса удаляется

.96. Примитивы синхронизации Windows. Понятие Dispatcher Object. Ожидание наступление события, вызовы Wait.

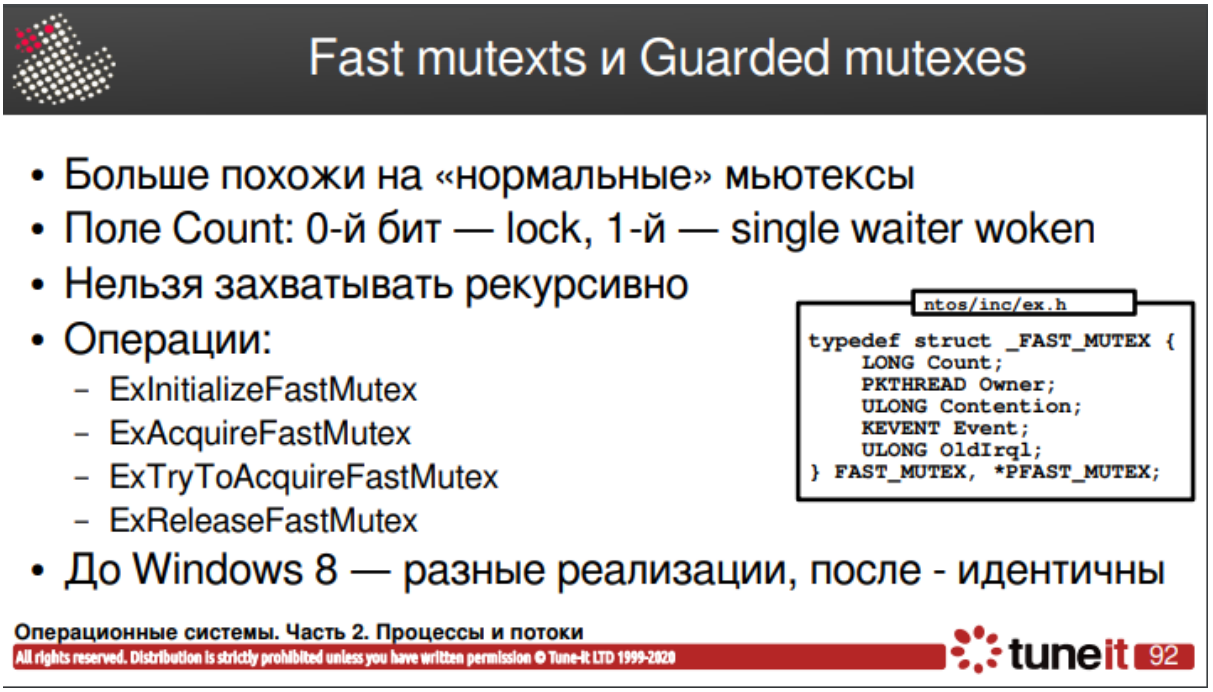




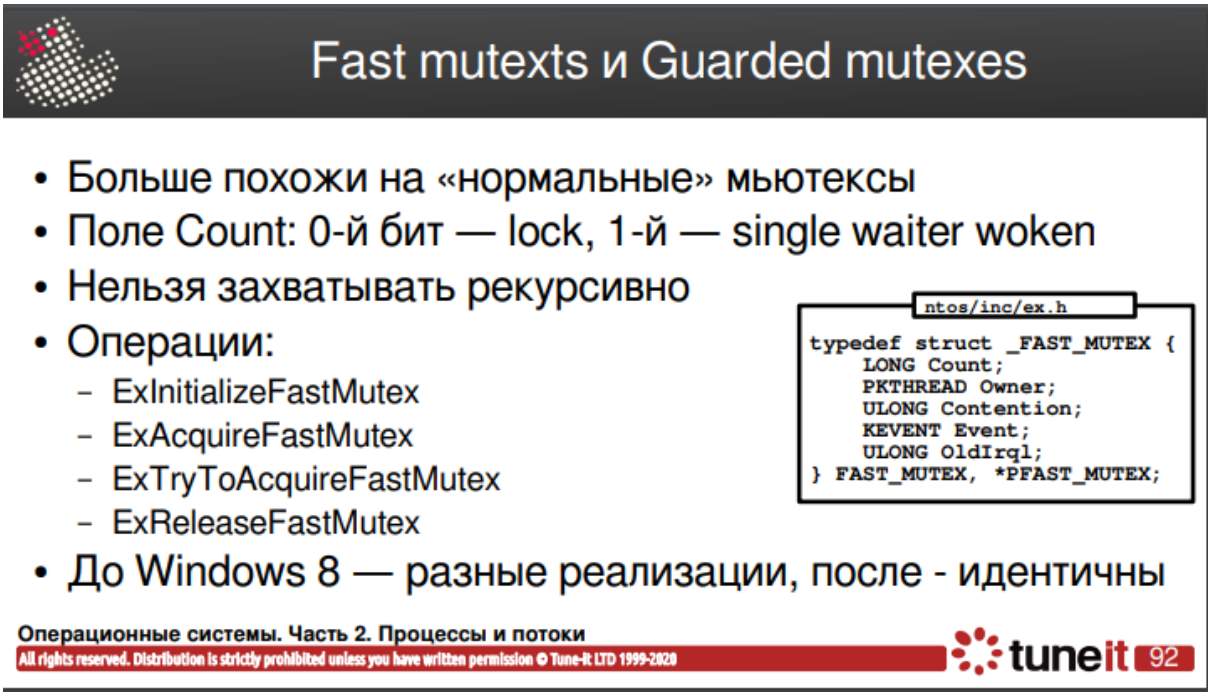
.97. Примитивы синхронизации Windows. EventObject, Mutex, Mutant.



.98. Примитивы синхронизации Windows. Fast mutex, Guarded mutex.



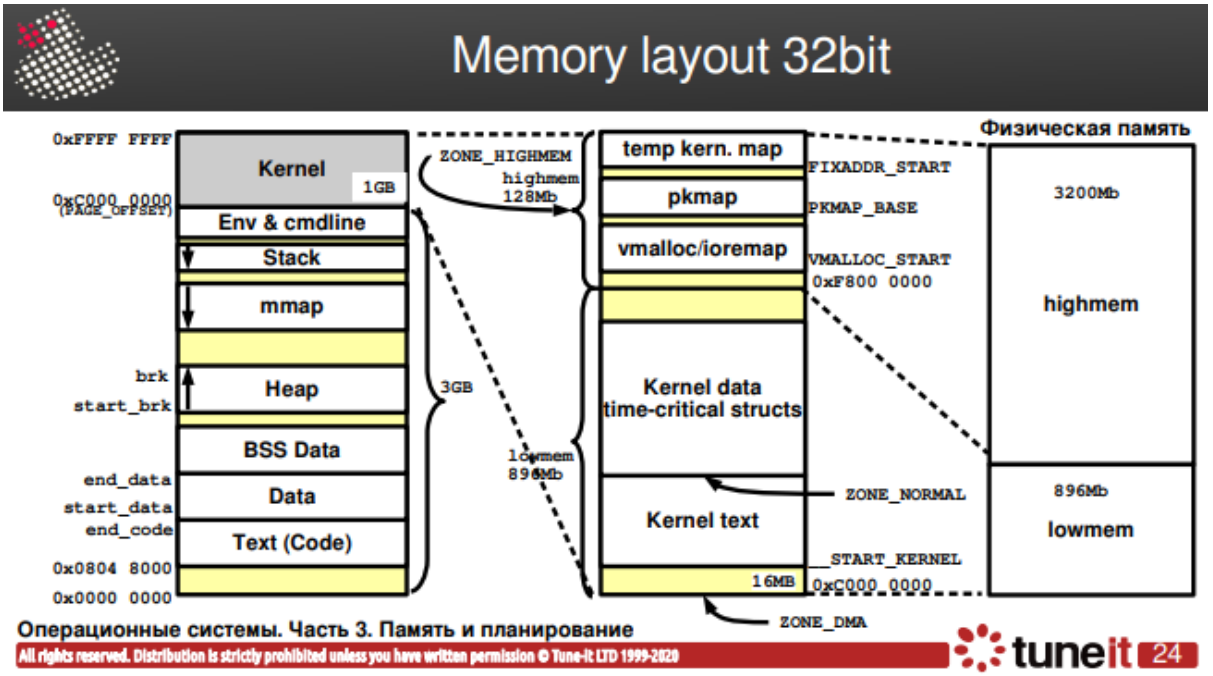
.99. Примитивы синхронизации Windows. Semaphore, spinlock



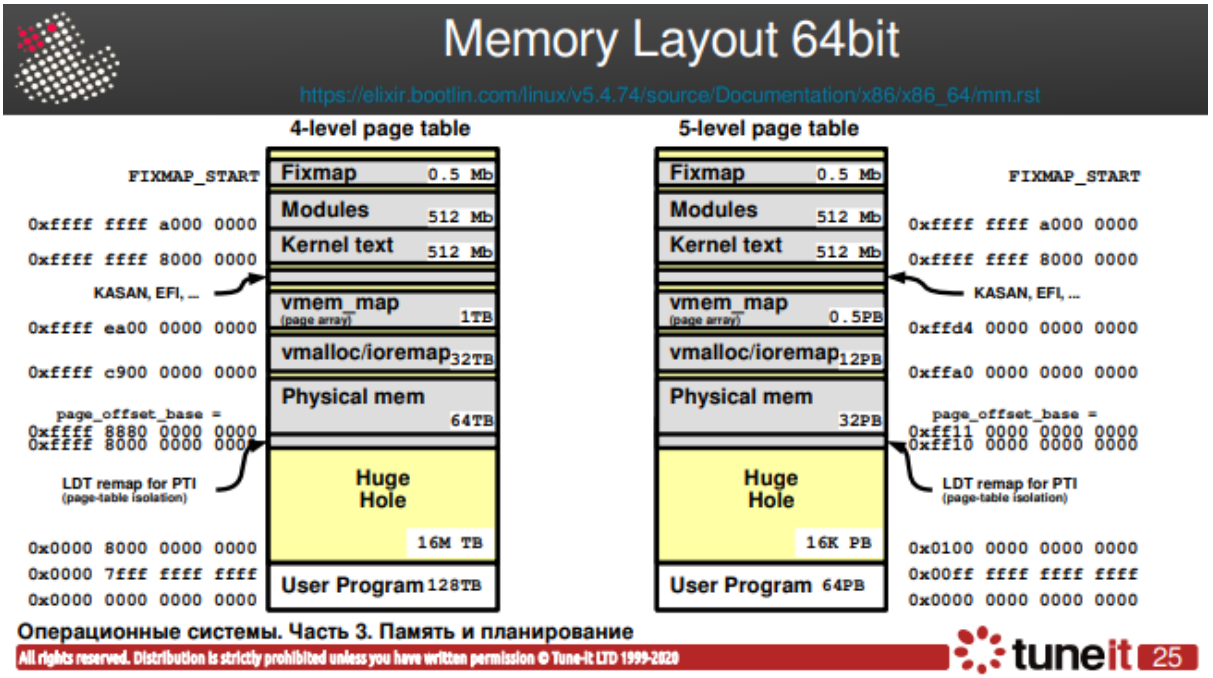
- .100. Хешированная таблица страниц SPARC64.

https://www.tune-it.ru/web/il/home/-/blogs/%D0%B2%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F-%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C-%D0%BD%D0%B0-sun-%D0%BA%D0%B0%D1%85-sparc64-%D0%B8-%D1%85%D1%8D%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0-%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%86

.101. Виртуальная память Linux. 32-х разрядная модель.



.102. Виртуальная память Linux. 64-х разрядные модели.



.103. Виртуальная память Linux. Структуры памяти.

Ничего не понятно. Худший билет во всем экзамене)

На слайде представлен кусочек того, что имеет отношение к структурам внутри памяти.

Начнем с тех частей, которые нам хорошо известны.

В первую очередь это task_struct (в левом нижнем углу схемы) - место, где расположена информация, относящаяся к процессу и находящаяся в памяти ядра. И это mm_struct, где находится описание памяти для процесса ядра.

Здесь выделено несколько переменных (первая из них указывает на mmap. Mmap – эта такая штука, которая представляет связанный список всех сегментов текущего процесса (предпоследний «столбец» справа)).

Причем видно, что каждый сегмент описывается структурой vm_area_struct. И в этой структуре находятся описательные элементы этого сегмента (например, есть адрес, где сегмент начинается и где заканчивается: vm_start, vm_end; есть права, указывающие на то, что разрешено делать в этой самой страничке: READ, EXEC; и есть указатели на следующий и предыдущий сегмент, чтобы в случае необходимости их можно было подвергнуть изучению).

В правом «столбце» в явном виде под каждым сегментом подписано, как бэкэндятся данные. Т.е. условно говоря, что «лежит под» этим самым сегментом. Имеется в виду, что сегменты «внизу себя» хранят либо своп-область и зарезервированное место в свопе, либо они хранят «внизу себя» файл.

И, соответственно, есть структура (struct_file), которая соответствует каждому сегменту, т. е. в явном виде есть ссылка на файл прямо из структуры vm_area_struct, где описано, что это за файл. И там есть поле f_mapping, которое указывает на маппинг файла.

Кроме того, есть анонимные странички Stack anonymous (показаны в верхнем правом углу схемы). У них есть отдельная структура, которая называется anon_vma, которая описывает анонимную область памяти.

Следует отметить, что сегментов в текущем процессе может быть очень много. Поэтому, для того чтобы найти сегмент по виртуальному адресу, существует красно-черное дерево поиска. Переменная в mm_struct (называемая mm_rb) ускоряет поиск по сегментам, необходимый для того, чтобы найти необходимый сегмент, которому принадлежит тот или иной адрес.

Таким образом, все сегменты, которые имеют «снизу» файл, имеют ссылку на struct_file. Все сегменты, которые анонимны - имеют структуру, соответствующую анонимной памяти.

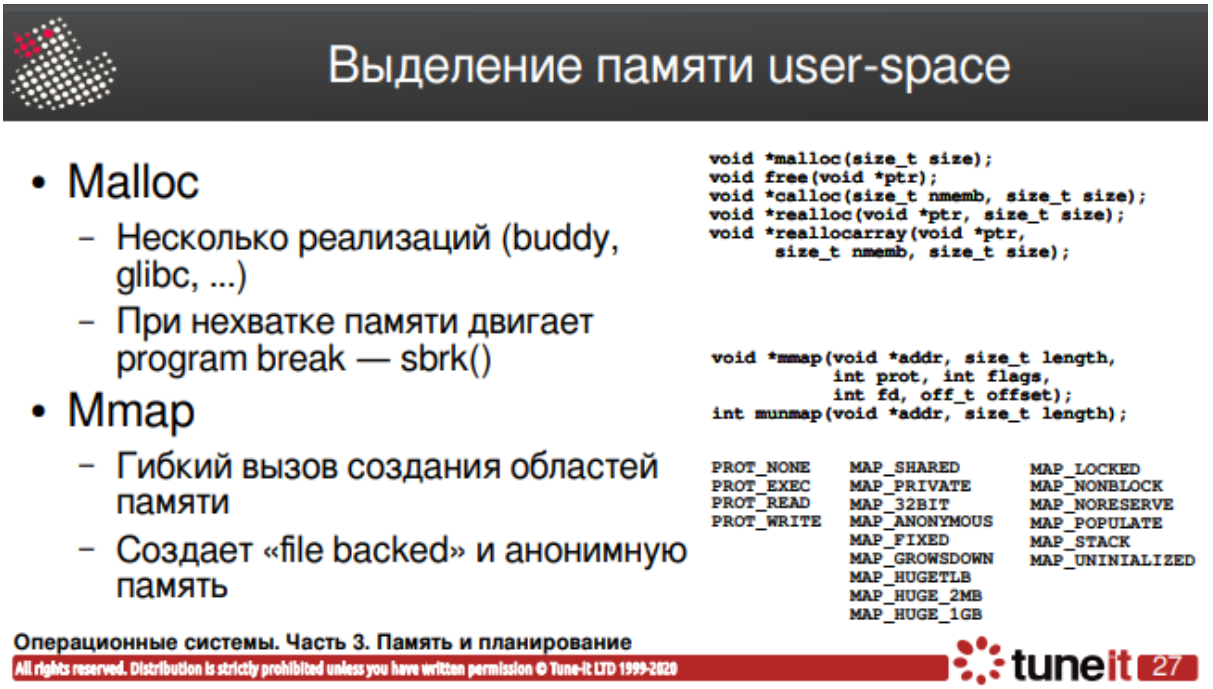
Странички «в регионе» struct_file могут или «поддерживаться» файлами, или быть анонимными. В каждой структуре памяти есть указатель на таблицу страниц. Table page «живет» относительно процесса: процесс при помощи регистра CR3 указывает на вход в нее; там есть специальные функции, которые осуществляют поиск по номеру странички. Поэтому легко можно найти в vmem_map (верхний левый угол слайда) соответствующую страничку просто прибавив к ней frame number.

В vmem_map хранятся описания всех страниц. Внутри каждой таблицы есть «реверс-маппинг», показывающий, кому она принадлежит. Так же там существует некоторое количество указателей, по которым можно определить, что находится в таблице. Поэтому, например, пэйджер и свопер, будет работать с каждой страничкой в зависимости от ее типа.

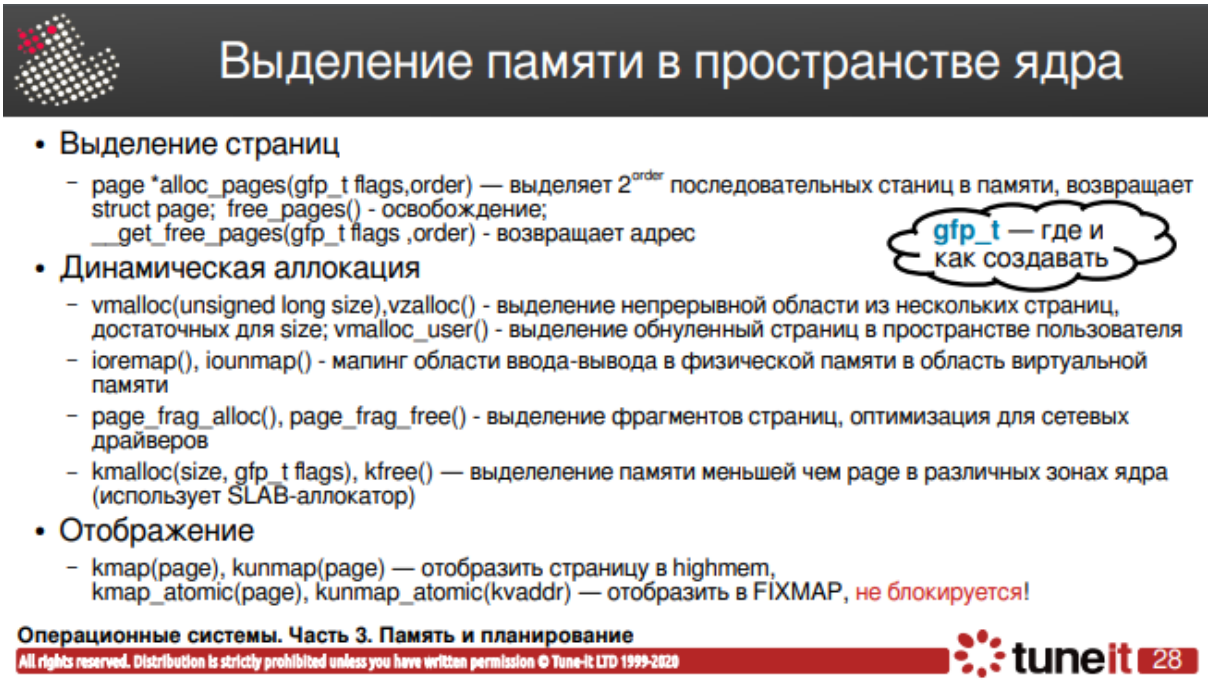
В address_space есть специальная переменная i_pages которая указывает на xarray (это тоже разновидность дерева для быстрого поиска). И этот xarray содержит странички, которые временно скэшированны в памяти. Т. е. описание всех страниц у нас есть. А в xarray будут у нас находиться те страницы, которые в реальности находятся в памяти и не требуют загрузки с диска.

Если нужно установить все элементы (страницы) файла, то для этого есть другая переменная, которая называется i_mmap. При этом тоже используется красно-черное дерево, которое устанавливает связи с блоками данных, которые являются файлом.

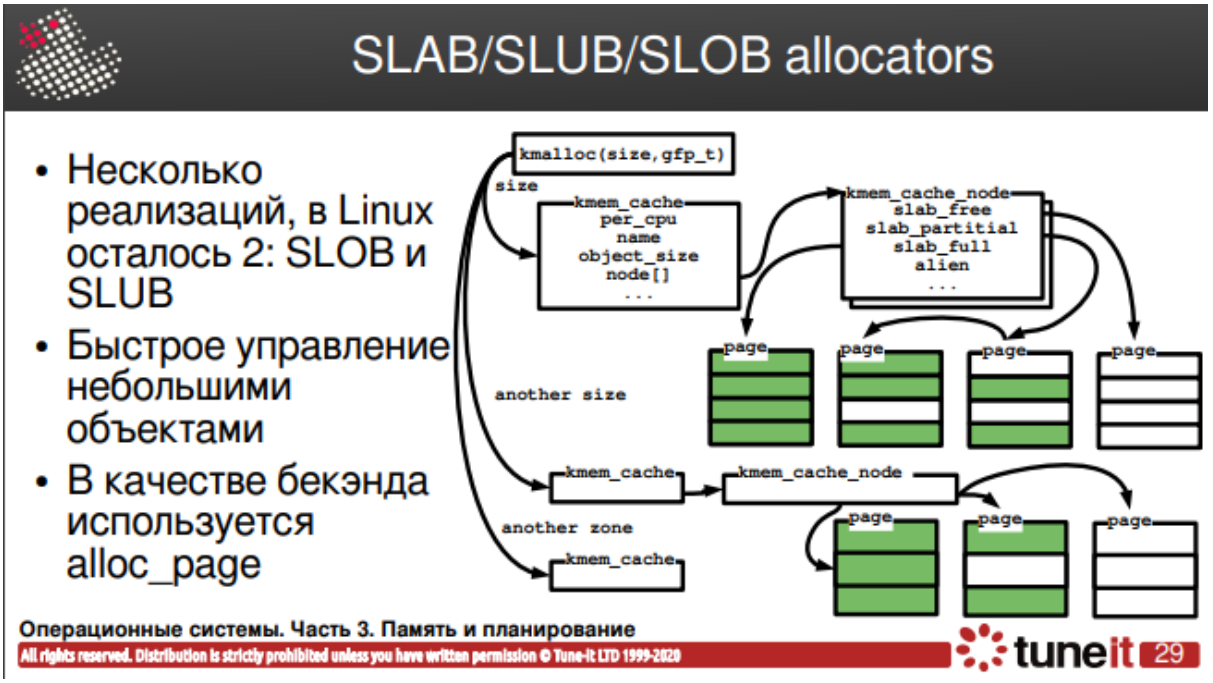
.104. Способы выделения памяти для пользовательских процессов Linux



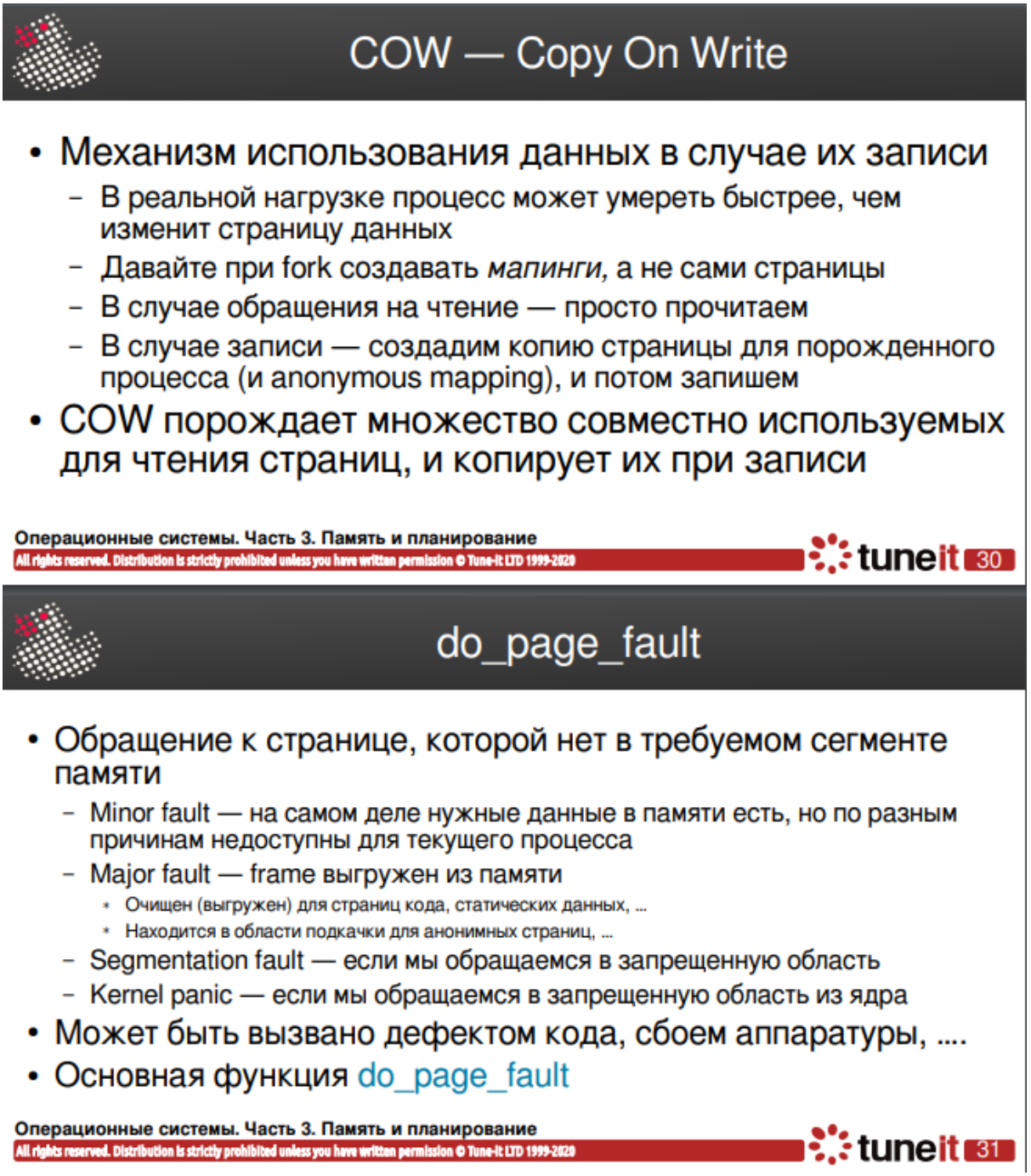
.105. Способы выделения памяти в пространстве ядра Linux.



.106. Слаб-аллокаторы SLAB/SLUB/SLOB.



.107. Copy on write и pagefault в Linux.



.108. Замещение страниц в Linux. Kswapd.