

Ссылочки:

Пример (оттуда в основном копипастим): [Пример](#)

Почти дубликат предыдущего: [OS. Экз](#)

- [1. Архитектура компьютерных систем. Архитектура Фон-Неймана и Гарвардская архитектура. Принципы архитектуры Фон-Неймана. Архитектуры NUMA и UMA.](#)
- [2. Общая организация процессора, памяти, организация вычислений.](#)
- [3. Организация прерываний, типы прерываний, контроллер прерываний.](#)
- [4. Типичные функции операционной системы. Интерфейсы ОС. Работа ОС как замена оператора ЭВМ.](#)
- [5. Пакетная обработка. Системный монитор.](#)
- [6. Анализ общесистемной эффективности, как предусловие многозадачности. Многозадачность, как способ повышения системной эффективности. Системы разделения времени.](#)
- [7. Процессы, проблемы современных процессов. Планирование выполнения процессов и управление ресурсами.](#)
- [8. Управление памятью, виртуальная память. Защита информации и безопасность ОС.](#)
- [9. Структура ядра операционной системы. Архитектуры монолитного ядра, ядра динамически загружаемыми модулями и микроядра.](#)
- [10. Потоки исполнения. многопоточность, модели многопоточности.](#)
- [11. Симметричная и асимметричная многопроцессорная обработка.](#)
- [12. Виртуализация. Типы виртуализации.](#)
- [13. Сбои и отказоустойчивость ОС. Причины появления отказов в ОС и способы борьбы с ними.](#)
- [14. Надежность. Среднее время восстановления. Коэффициент доступности и время простоя.](#)
- [15. Резервирование и отказоустойчивость.](#)
- [16. История и развитие ОС GNU/Linux. Single UNIX Specification и POSIX.](#)
- [17. Понятие дистрибутива, дистрибутивы Linux.](#)
- [18. Архитектура и основные подсистемы Linux. Linux Kernel Map.](#)
- [19. История и развитие Windows](#)
- [20. Общая архитектура Windows. Windows API](#)
- [21. Сервисы, функции и важные компоненты Windows.](#)
- [22. Процесс, характеристики процесса в момент выполнения. Состояние процесса. Разделение ресурсов.](#)
- [23. Модель процесса с пятью состояниями, назначение состояний.](#)
- [24. Paging и Swapping. Модель процесса с семью состояниями.](#)
- [25. Управляющие таблицы процесса. Образ процесса.](#)

- [26. Управляющий блок процесса \(PCB\). состав PCB](#)
- [27. Функции ОС, связанные с процессами. Создание процесса, переключение процессов.](#)
- [28. Процессы в ОС UNIX SVR4. Диаграмма состояний, основные структуры.](#)
- [29. Понятие потока выполнения. связь потока и процесса. Преимущества потоков.](#)
- [30. Состояния потока, User Level Threads vs Kernel Level Threads](#)
- [31. Многопроцессорность и многопоточность. Закон Амдала.](#)
- [32. Механизм параллельных вычислений, функции ОС.](#)
- [33. Проблемы параллельного выполнения: взаимоисключения, взаимоблокировки, голодание. Требования к взаимным исключением. Уровни взаимодействия процессов и потоков.](#)
- [34. Примитивы синхронизации ОС. Предназначение примитивов синхронизации](#)
- [35. Примитивы синхронизации ОС. Семафоры и мьютексы. Бинарный семафор](#)
- [36. Примитивы синхронизации ОС. Условные переменные, rwlocks.](#)
- [37. Примитивы синхронизации ОС. Мониторы, флаги событий, передача сообщений.](#)
- [38. Примитивы синхронизации ОС. Неблокирующие примитивы синхронизации и неблокирующие структуры данных.](#)
- [39. Управление памятью, основные определения и требования к организации.](#)
- [40. Фиксированное и динамическое размещение программ в памяти.](#)
- [41. Модели аппаратного перемещения программ.](#)
- [42. Простой страничный поход и простая сегментная организация.](#)
- [43. Виртуальная память основные определения и принципы организации аппаратуры и управляемых программ.](#)
- [44. Виртуальный страничный обмен. Двухуровневая организация MMU и TLB 80386](#)
- [45. Инвертированная таблица страниц.](#)
- [46. Сегментно-страничная виртуальная память.](#)
- [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.](#)
- [49. Виды планирования процессов. Критерии краткосрочного планирования. Приоритеты.](#)
- [50. Использование приоритетов.](#)
- [51. Стратегии планирования FCFS, RR, SPN, SRT, HRRN, Feedback.](#)
- [52. Feedback планировщик и классы планирования ОС UNIX SVR4.](#)
- [53. Справедливое планирование.](#)

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

[55. ОС реального времени и планировщики. Deadline-планирование.](#)

[56. Проблема инверсии приоритетов, типы инверсии и способы решения в планировщике.](#)

[57. Ввод-вывод. Современные устройства и скорости обмена, развитие способов ввода-вывода, логическая структура ввода-вывода.](#)

[58. Буферизация ввода вывода. Ввод-вывод в UNIX SVR4.](#)

[59. Диски и дисковое планирование.](#)

[60. Концепции RAID.](#)

[61. RAID-0, 1, 10, 0+1.](#)

[62. RAID 4,5,6. Аппаратные дисковые массивы.](#)

[63. Файловый ввод-вывод, основные определения. Задачи ОС по управлению файлами. Совместное использование файлов.](#)

[64. Управление файлами в UNIX SVR4](#)

[65. Каталоги файлов. Элементы каталога, операции ОС.](#)

[66. Размещение записей и файлов в блоках данных. Сложность и типы организации размещения.](#)

[67. Непрерывное размещение файлов \(на примере ОС RT-11\)](#)

[68. Цепочечное размещение файлов \(на примере DOS FAT\)](#)

[69. Индексированное размещение \(на примере файловой системы UNIX UFS\)](#)

[70. Linux: стандартные средства для наблюдения счетчиков ядра.](#)

[71. Linux: файловая система /proc.](#)

[72. Linux: трассировщики системных вызовов и библиотек.](#)

[73. Linux: Профилировщик perf и FlameGraph.](#)

[74. Linux: SystemTap.](#)

[75. Linux: Отладчик ядра.](#)

[76. Windows: стандартные отладочные средства.](#)

[77. Windows: утилиты SysInternals](#)

[78. Windows: отладчики WinDbg и KD](#)

[79. Аппаратная поддержка взаимных исключений.](#)

[80. Эволюция подхода к блокировке \(Столлингс, гл. 5.1\)](#)

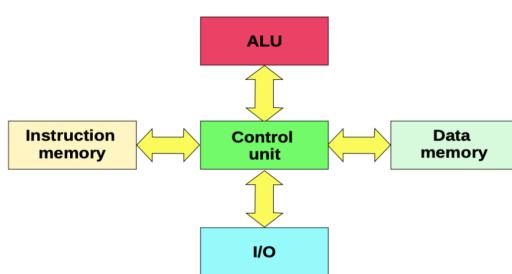
[81. Принципы взаимного блокирования \(Столлингс, гл. 6.1\)](#)

[82. Предотвращения взаимоблокировок, устранение взаимоблокировок, обнаружение блокировок. \(Столлингс, гл. 6.2, 6.3, 6.4\)](#)

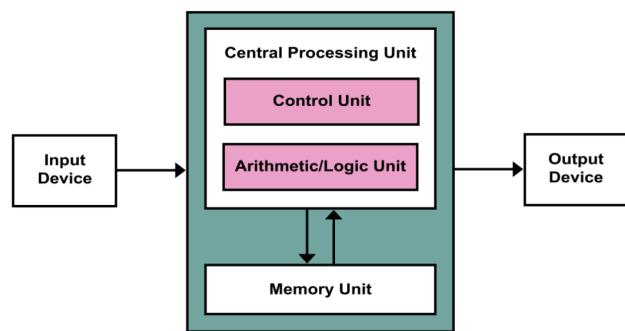
- [83. Задача об обедающих философах \(Столлингс, гл. 6.6\)](#)
- [84. Процессы в Linux: структура task_struct, поля структуры, связь с другими структурами ядра.](#)
- [85. Диаграмма состояния процесса Linux.](#)
- [86. Создание процесса Linux на уровне пользовательского процесса.](#)
- [87. Создание и завершение процесса Linux на уровне ядра. Вызываемые функции.](#)
- [88. Особенности реализации потоков в Linux. KThread, Tasklet.](#)
- [89. Примитивы синхронизации Linux. Spinlock и qspinlock.](#)
- [90. Примитивы синхронизации Linux. Semaphore и Mutex.](#)
- [91. Примитивы синхронизации Linux. rw_semaphore, seqlock.](#)
- [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.](#)
- [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.](#)
- [101. Виртуальная память Linux. 32-х разрядная модель.](#)
- [102. Виртуальная память Linux. 64-х разрядные модели.](#)
- [103. Виртуальная память Linux. Структуры памяти.](#)
- [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.](#)

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

Гарвардская архитектура



Архитектура фон Неймана



Два основных вида: **Гарвардская и фон Неймана.**

Гарвардская: отдельная память для команд, отдельная для данных; отдельные каналы для них.

Фон Нейман: данные и команды хранятся в одной и той же памяти.

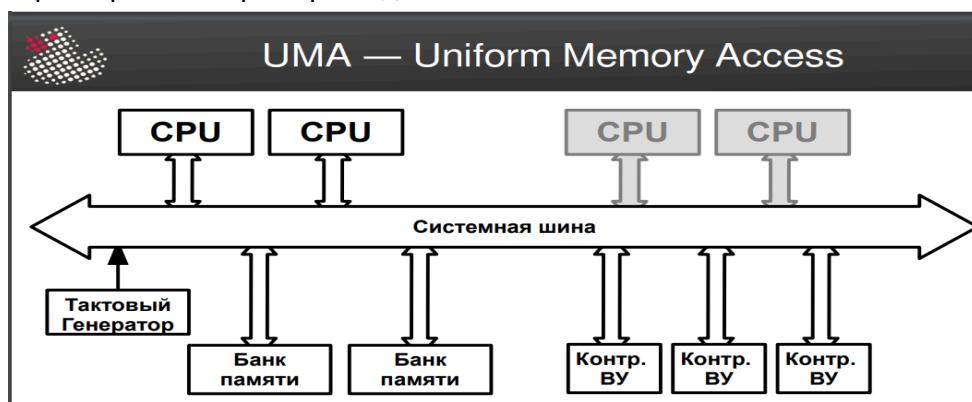
Фон Нейман проще, но общая шина памяти становится узким местом.

Принципы Фон-Неймана:

- однородность памяти (команды и данные хранятся вместе, внешне неразличимы);
- адресность (у каждой ячейки памяти есть адрес, по которому обращается процессор);
- программное управление (вычисления в виде программы из последовательности команд);
- двоичное кодирование (данные и команды кодируются 0 и 1).

Схемы реализации компьютерной памяти UMA и NUMA:

- **UMA** (Uniform Memory Access / Однородный доступ к памяти):
Системная шина - общий канал обмена информацией.
Одноранговость - все устройства подключены к системнойшине.
Единообразный доступ к памяти -> время доступа, задержка и другие сист. характеристики примерно одинаковы.



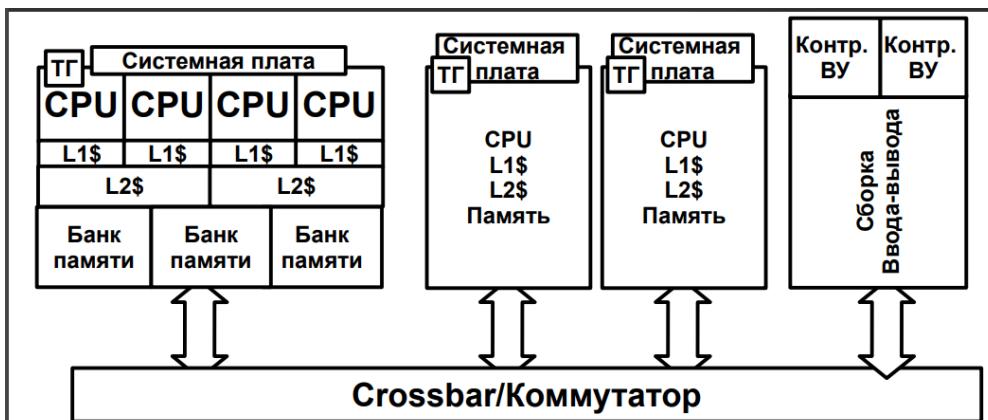
Плюсы:

- Одинаковое время доступа к памяти
- Простота реализации

Минусы:

- Плохая горизонтальная масштабируемость (>2 CPU - конкуренция за доступ к памяти, шина блокируется (узкое место), система замедляется)

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



- Системные ресурсы (кэши, процессоры и банки памяти) собраны в системные платы
- Коммутатор - блок для полносвязной архитектуры (соединяет все порты) вместо системной шины (исчезает узкое место)
- Быстрее доступ к памяти на локальной плате, чем на других (affinity)
- Используется в серверах

Плюсы:

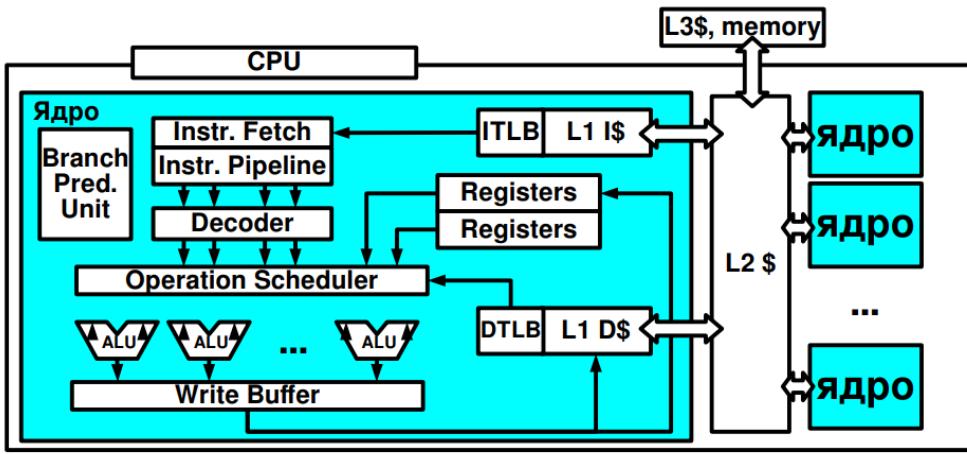
- Возможность горячей замены компонентов
- Хорошая горизонтальная масштабируемость (больше процессоров)
- Надежность и отказоустойчивость
- Можно поставить разную частоту ТГ для разных плат

Минусы:

- Требования к ОС (она должна поддерживать все эти фишки)
- Нужно учитывать affinity
- Сложность по сравнению с UMA

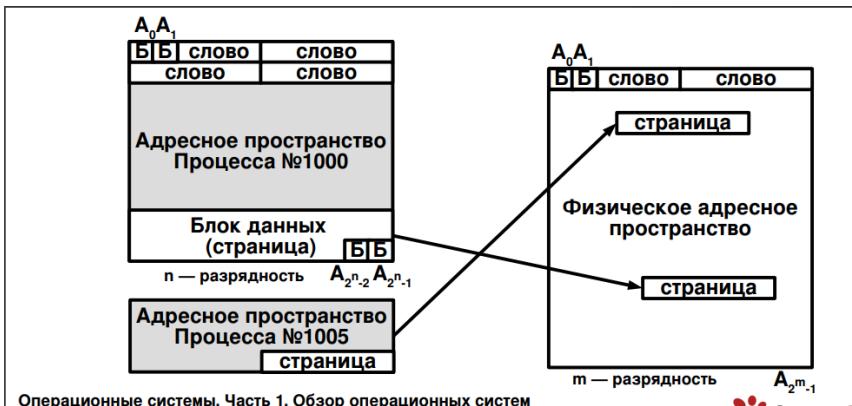
2. Общая организация процессора, памяти, организация вычислений.

Организация процессора



- Процессор состоит из отдельных ядер
- Кэш второго уровня (L2\$): соединен с памятью, L3\$ и частями L1\$ каждого ядра
- Кэш первого уровня (L1\$): разделен на кэш с данными (D\$) и инструкциями (I\$) с собственными TLB, это помогает параллельно читать данные и инструкции (а-ля Гарвард)
- Буфер ассоциативной трансляции ([TLB](#)): хранит недавние маппинги виртуальных и физических адресов, ускоряя их трансляцию и поиск данных в кэше
- Многопоточность (Registers): у процессора есть два набора регистров, что позволяет быстрее переключать контекст двух потоков
- [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 (Instr. Fetch), декодирует (Decoder) и отдает планировщику операций (Operation Scheduler)
 - планировщик решает, пустить пришедшие команды параллельно или нет и пустить ли вообще команду на выполнение, в зависимости от наличия данных
- Предсказатель переходов (Branch Prediction Unit, BPU): на основании статистики предсказывает выполняемую ветку условия и отдает приказ на предвар. загрузку и исполнение (позволяет избежать сброса конвейера и простоя процессора)
- Write Buffer - буферизация результатов перед записью в память, работающей медленнее исполнения команд
- АЛУ, обычные и специфичные (для плавающей точки, для векторных операций и т.п.)

Организация памяти



- Виртуальная память – способ организации памяти, при котором каждый процесс имеет собственное АП, 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) и большие (Large) страницы.

Преимущества: 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 (поэтому малая) память наверху (ближе к процессору), снизу самая большая, медленная и дешёвая. Сделано так, чтобы к более медленной памяти было меньше обращений.

Тд (время доступа) – интервал между первым запросом доступа к памяти и поступлением первых данных.

Типы памяти:

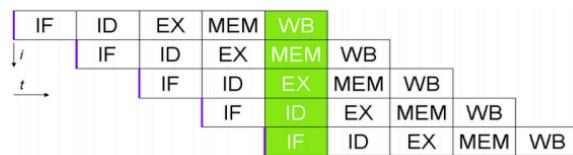
- Регистровая - вид памяти, модули которой содержат регистр между микросхемами памяти и системным контроллером памяти
- Ассоциативная - адресация на основе содержания, а не местоположения данных -> ускорение поиска необходимых записей
- Адресная - адресация по адресу (а не по содержанию, как в ассоциативной)
- Блочная - объединение памяти в блоки. Физический адрес ячейки вычисляется по номеру блока и непосредственно адресу ячейки внутри этого блока, читается блок целиком
- Последовательная - только последовательный доступ (как в связном списке)

Виды управления:

- Компилятор: управляет при помощи ключей компиляции (уровни оптимизации -O0, -O1, ...)
- Аппаратура: поиск при помощи схем в ассоциативной памяти (TLB)
- Программно

Организация вычисления

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



Параллельность -> Разнесение циклов по времени -> Ускорение процессора

Следующая команда выполняется до окончания выполнения предыдущей

Требуется минимизировать сбросы очереди и загрузку с нуля, поэтому загружаются обе ветви или предсказываются.

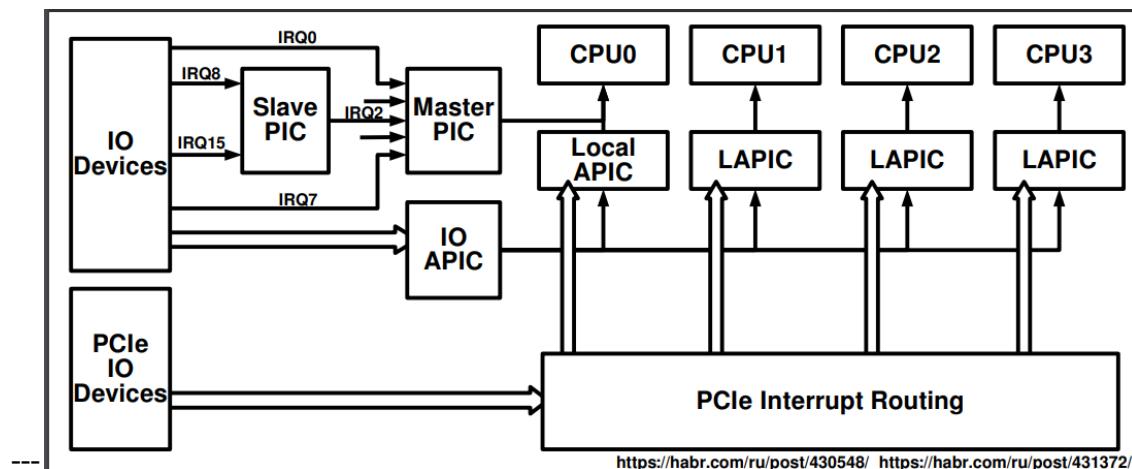
3. Организация прерываний, типы прерываний, контроллер прерываний.

Прерывание - 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
- Выполняются в конце цикла команды
- Могут быть вложенными. Есть ограничения по вложенности обработчика в зависимости от архитектуры. Прервать обработчик прерывания может обработчик прерывания с более высоким приоритетом

Контроллер прерываний (англ. Programmable Interrupt Controller, PIC) — функциональный блок, отвечающий за возможность последовательной обработки запросов на прерывание от разных устройств, а также за возможность подключать больше устройств.



Надо читать всю статью с [хабра](#)

Слайд интерпретируется так:

1. Slave-Master PIC - были придуманы для увеличения числа линий прерываний (выход slave подключается к одному из входов master)
 2. IO APIC, LAPIC (Advanced PIC) - был придуман для поддержки многопроцессорности (LAPIC на каждом ядре, IO APIC балансирует нагрузку между разными LAPIC)
 3. (PCIe = Peripheral Component Interconnect Express) PCIe_IO_Device -> PCIe_Interrupt_Routing -> Local_APIC -> CPU - пришло с появлением PCIe, информация о прерывании пишется (почти) напрямую в MMIO-область памяти LAPIC. PCIe_Interrupt_Routing - маршрутизатор прерываний. Также называется MSI (memory signaled interrupts)
-

Разные подходы:

-запрет повторных прерываний, пока обрабатываются предыдущие (последовательная обработка прерываний).

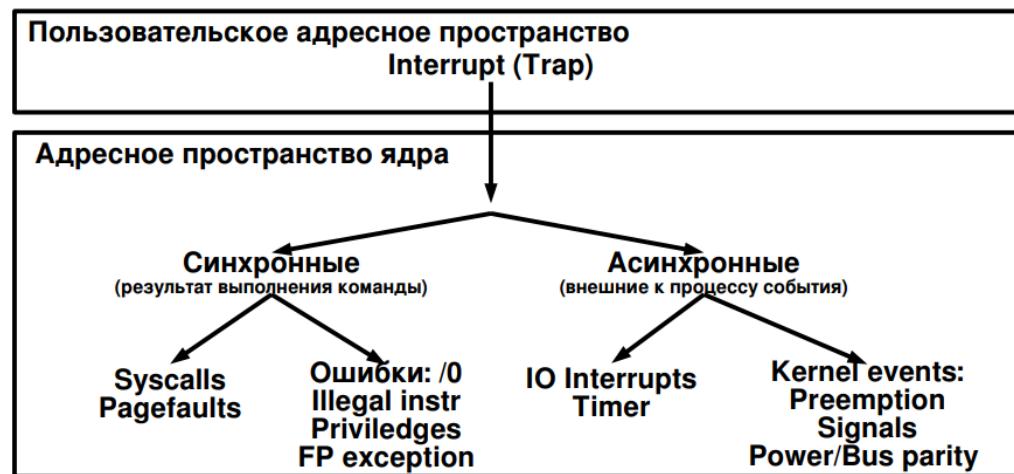
-обработка прерываний по уровню приоритета

(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 посыпает процессору запрос на прерывание.

ТИПЫ ПРЕРЫВАНИЙ:



Асинхронные, или внешние к процессу - 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 (Interrupt Request, IRQ) - устройства сообщают, что они требуют внимания со стороны ОС; также внешними являются различные события ядра: вытеснение процесса, сигналы и т.д.

Синхронные, или внутренние, как результат выполнения команды - 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, или как 0x80h (syscall), который осуществляет системный вызов ОС.

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 интерфейсов 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.

Контролируемый доступ к файлам состоит в том, что ОС разделяет пользователей на классы и наделяет их различными правами.

Обнаружение и обработка ошибок - дампинг памяти и запись в лог данных об ошибке при возникновении исключительных ситуаций.

Диспетчеризация ресурсов - функция ОС, позволяющая программам разделять общие ресурсы путем назначения им необходимых прав.

API: Application Program Interface

This is the set of public types/variables/functions that you expose from your application/library.

In C/C++ this is what you expose in the header files that you ship with the application.

ABI: Application Binary Interface

This is how the compiler builds an application.

It defines things (but is not limited to):

- How parameters are passed to functions (registers/stack).
- Who cleans parameters from the stack (caller/callee).
- Where the return value is placed for return.
- How exceptions propagate.

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 внимание на то, что ISA определяет набор команд, а не способ их выполнения.

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.

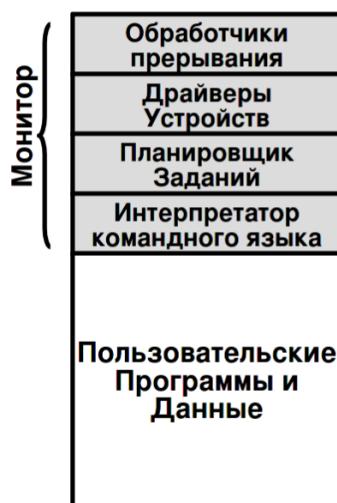
До оператора ученые = программисты сами набирали программу, запускали и уходили на какое-то время, пока она не выполнит свою работу. Было расписание, по которому пользователи приходили со своими программами.

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.

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



Цель: нивелирование минусов последовательной обработки (расписание, подготовка программы к исполнению).

Первая ОС (в т.ч. пакетная) - середина 1950-х, компания General Motors, компьютер IBM 701.

Программист готовит программы заранее на перфокартах/магнитных лентах, оператор их собирает в пакеты, помещает в устройство ввода данных и запускает.

Монитор считывает с УВД по одному заданию, оно размещается в области памяти программ пользователей, получает управление и по завершении или сбою возвращает его монитору, запускающему следующую. Результат исполнения печатается на устройство вывода.

Резидентный монитор - большая часть монитора, всегда в основной памяти и готова к работе (сост. части на слайде выше).

Оставшаяся часть - утилиты и общие функции, вызываемые при необходимости в начале выполнения задания.

Также есть **JCL** (язык управления заданиями), помогающий в подготовке программы к исполнению (указать компилятор Фортрана - **\$FTN** или запись на магнитную ленту - **\$LOAD**).

Итог: по сути весь доп. софт для работы программы заранее загрузили в ЭВМ, плюс добавили прерывания, чтобы сигнализировать о завершении программы, и планировщик, чтобы запускать следующую программу. Программы все еще выполняются последовательно и надо долбиться с их загрузкой, но можно сэкономить время за счет отказа от издержек расписания, возможности загрузить несколько программ сразу и более гибкой подготовки к исполнению.

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



Обще-системная эффективность

- 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



Многозадачность



IO работает намного медленнее CPU (здесь - в 15 раз) -> Процессор в течение цикла очень слабо загружается (например, если бы мы печатали огромный отчёт, то CPU бы до конца простоявал).

Можно запустить несколько задач, разбить на кусочки и переключать по окончании вычислительной части.

Здесь “ожидание” - не только ожидание ресурсов, но и IO. Его время сократится за счет эффективного распределения заданий по системе с учётом того, что процессор тратит время и на вычисления, и на IO.



Системы разделения времени (Time Sharing)

- 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 (Compatible Time-Sharing System), MIT 1961, IBM 709
 - Выгрузка и загрузка задач
 - 32 пользователя
- Появились проблемы разделения ресурсов и защиты одних программ от других

Time Sharing - совместное использование ресурса (а не “деление”).

Основная идея - распределение между пользователями за счет квантования времени; каждому даётся лишь 1/n раб. времени, но при оптимальном размере группы и хорошей настройке системы время отклика сравнимо с реакцией пользователя.

Отдельный пользователь использует компьютер неэффективно, но группа пользователей (оптимального размера), работая совместно, резко повышает эффективность работы (например, при вводе-выводе с клавиатурой, диска или ленты, или передаче по сети).

Использование ресурсов внутри системы разными процессами → Исключение оператора, добавление пользователей (которые сами выполняют его работу) и большого количества терминалов.

CTSS - первая ОС общего назначения с разделением времени (возможность выгрузки и загрузки задач, 32 юзера). Таймер каждые 0,2 с генерирует прерывание, передавая управление ОС и другому юзеру, делая дамп данных и программы старого.

Примитивный, но эффективный метод -> монитор минимальных размеров (5000).

Задание грузится в одно и то же место в памяти (после монитора) -> нет необходимости в методах перемещения.

Появились проблемы разделения ресурсов и защиты одних программ от других (вирт. памяти не было -> можно было залезть в чужую задачу).

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



Процессы

- Multics, 1965 г. General Electric, Bell Labs.
- Процесс — совокупность взаимосвязанных и взаимодействующих операций, преобразующих входящие данные в исходящие. (ISO 9000:2000)
- Процесс — экземпляр программы во время ее исполнения
- Процесс — единица активности ОС, в которой существуют последовательные действия, текущее состояние и набор связанных ресурсов.

Три определения - ISO, пользователь и ОС.

- Исполняемая программа
- Набор потоков исполнения
- Связанные структуры ядра
- Адресное пространство
 - Код, данные, стек, куча
- Контекст исполнения
- Контекст безопасности
- Ресурсы (файлы и пр.)
- Динамические библиотеки



Поток исполнения - единица потребления мощности процессора.

Связанные структуры ядра - описание ресурсов, которые содержит процесс (task_struct) - linux.

Адресное пространство - на слайде:

- Сегмент кода - содержит набор машинных команд, предст. собой скомпилированную программу пользователя. Можно только читать и исполнять (r-x).
- Сегмент данных - можно только читать и менять (rw-).
- Сегмент “куча”.
- Стек - адреса возврата и параметры передаваемых функций.
- Библиотеки - линкуемые статически (при компиляции) и динамически (после компиляции).
- Ядерная часть

Контекст исполнения - набор регистров РСВ, содержащий состояние программы.

Контекст безопасности - идентификаторы (группы, пользователя, эффективный и т.д.).

Ресурсы - открытые файлы, сетевые соединения и т.д.



Проблемы современных процессов

- Защита памяти процессов
 - Недетерминированное поведение программы
- Взаимные блокировки
 - deadlocks, starvation, livelocks
- Проблемы синхронизации
- Взаимное исключение доступа к ресурсам

Недетерминированное поведение программы.

Результат работы каждой программы обычно должен зависеть только от ее ввода и не должен зависеть от работы других программ, выполняющихся в этой же системе.

Однако в условиях совместного использования памяти и процессора программы могут влиять на работу друг друга, переписывая общие области памяти непредсказуемым образом. При этом результат работы программ может зависеть от порядка, в котором они выполняются.

Взаимоблокировки.

Современные ОС и программы очень сильно подвержены некорректным алгоритмическим решениям программистов и, как следствие, созданию условий для появления взаимных блокировок на уровне как процессов, так и потоков. Наиболее часто встречающиеся разновидности блокировок: deadlock, starvation и livelock.

Deadlock - это взаимное блокирование процессов друг другом, когда один процесс пытается заполучить ресурс, которым владеет другой процесс, а тот в свою очередь владеет первым. Это может происходить при использовании мьютексов, файлов и др. Процессы не могут продвинуться дальше в своих вычислениях, взаимно блокируя друг друга, находясь в ожидании освобождения блокировки.

Livelock - похож на deadlock, с той лишь разницей, что оба конкурирующих процесса, находясь в состоянии исполнения на процессоре, одновременно циклически проверяют доступ к ресурсу, при этом оба на 100% загружают CPU.

Starvation - ситуация, при которой поток не блокируется, но в течение длительного времени недополучает ресурсов. Для решения этой проблемы ОС реализуют наследование приоритетов и используют сложные алгоритмы планирования.

Неправильная синхронизация.

Часто случается так, что программа должна приостановить свою работу и ожидать наступления какого-то события в системе.

Например, программа, которая инициировала операцию ввода-вывода, не сможет продолжать работу, пока в буфере не будут доступны необходимые ей данные. В этом случае требуется передача сигнала от какой-то другой программы. Недостаточная надежность сигнального механизма может привести к тому, что сигнал будет потерян или будет получено два таких сигнала.

Сбой взаимного исключения.

Часто один и тот же совместно используемый ресурс одновременно пытаются использовать несколько пользователей или несколько программ. Например, два пользователя могут попытаться одновременно редактировать один и тот же файл. Если эти обращения не контролируются должным образом, возможно возникновение ошибок. Для корректной работы требуется некоторый механизм взаимного исключения, позволяющий в каждый момент времени выполнять обновление файла только одной программе. Правильность реализации такого взаимного исключения при всех возможных последовательностях событий крайне трудно проверить.



Планирование выполнения процессов и управление ресурсами

- Равноправие
 - Пользователи должны получать ресурсы равноправно
- Дифференциация отклика
 - В некоторых задачах нужно понизить время отклика
- Общесистемная эффективность
- Планировщики процессов, дисков и пр.
 - Разные классы диспетчеризации (Time Sharing, Interactive, Real Time, System, Fair Share, Fixed...)

Равноправие - ОС полагает, что юзеры, процессы и объекты, потребляющие ресурсы, должны быть равноправны по отношению к остальным, их права по умолчанию должны ничем не отличаться, а доступ к ресурсам делиться между ними в равной пропорции. Достигается за счёт использования планировщиков.

Пример к дифференциации отклика: в UNIX, если окно находится в фокусе, система поднимает его приоритет, т.к. пользователь при работе с интерактивным приложением должен получать отклик на свои действия без заметных задержек.

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



Управление памятью

- Изоляция процессов
- Управление выделением и освобождением памяти
 - Heap allocator, Kernel allocator, mapping files
- Поддержка модулей
 - Динамическая загрузка модулей
- Защита и контроль доступа
 - Права на сегменты памяти (Н-р: поexec data, stack)
- Долговременное хранение
- Страницочный обмен
 - Paging, swapping

В современных ОС под управлением памятью понимается:

1. Изоляция процессов (отслеживание, что ни один из процессов не смог изменить чужую память). Для этого каждый процесс работает в своей песочнице (виртуальном АП).
2. Управление выделением и освобождением памяти (программы должны динамически подключаться к памяти - т.е. надо в том числе уметь передать ей управление). Heap, kernel allocator, mapping files
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 (вторичная память - диски, облака и т.д.).
6. Страницочный обмен (**Paging, swapping**) - выгрузка фрагментов памяти из ОЗУ во вторичную память.



Виртуальная память

- Отдельное виртуальное адресное пространство для каждого процесса и ядра
- Использование подкачки страниц с диска для эффективного использования памяти
 - «Увеличение доступной памяти»
- Управление MMU и TLB
- Невыгружаемые страницы

Виртуальная память – схема распределения памяти, при которой АП каждого процесса организовано с помощью одинакового набора виртуальных адресов. При этом физические адреса уникальны, а процессы изолированы и могут обращаться друг другу только при помощи спец. вызовов (например, получение доступа к shared memory).

Чаще всего виртуальная память это набор таблиц страниц, которые должны отображать все доступное АП машины. Доступ к определенной странице - последовательно идущие id таблиц страниц, и в конце смещение до нужного адреса. В современных ОС виртуальная память реализуется программно-аппаратным комплексом. Благодаря пейджингу и своппингу мы можем выгружать неиспользуемые страницы из первичной памяти во вторичную. При последующем обращении к странице ОС видит, что страницы нет в памяти и генерирует Page Fault (сигнал страницной ошибки).

Алгоритм поиска страницы - сначала просматриваем TLB, если там нет, ищем в ОЗУ, если оказывается, что страницы нет и в ОЗУ, то ее необходимо достать из вторичной памяти.

Изменяемые в процессе работы данные (например, страницы из кучи) выгружаются в область подкачки, но неизменяемые (например, сегменты кода) можно просто уничтожить и потом загрузить при необходимости из исполняемого файла.

За счет расходов на пересылку страниц мы можем расширить доступную для процессов память, но некоторые страницы в любом случае надо держать резидентно (части кода ядра, часто используемые разделяемые библиотеки), потому что их выгрузка может сильно снизить скорость работы многих программ.

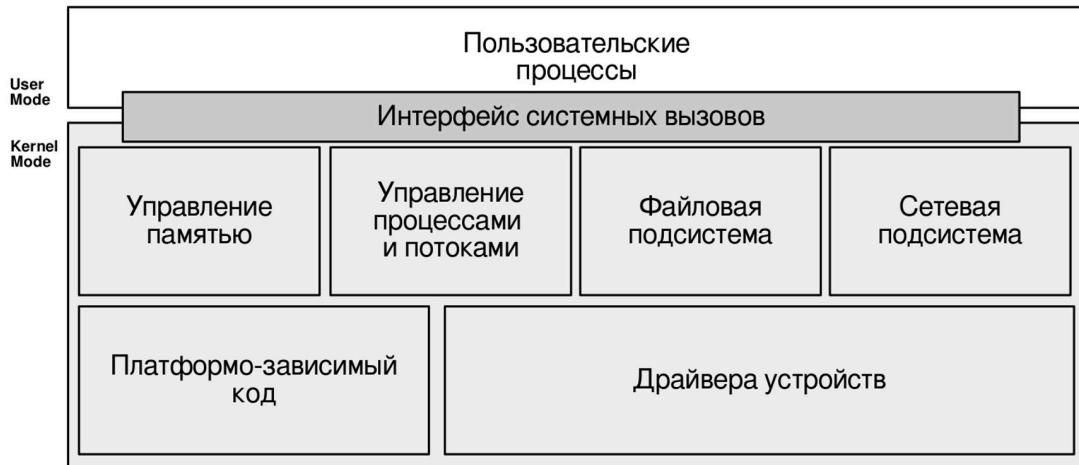


- Доступ к системе
 - Защита от несанкционированного доступа
- Конфиденциальность
 - Невозможность неавторизованного доступа к данным
- Целостность данных
 - Защита данных от неавторизованного и нецелостного изменения
- Аутентификация и авторизация

- Доступ к системе (защита от несанкционированного доступа)
- Конфиденциальность (необходимость предотвращения утечки некоторой доверительной информации)
- Целостность данных (защита данных от неавторизованного и нецелостного изменения, т.е. они должны оставаться неизменными при отображении, передаче, хранении и т.д.)
- Аутентификация и авторизация
 - **Аутентификация** — процедура проверки подлинности личности, например, путем сравнения введенного пароля с сохраненными в БД.
 - **Авторизация** — проверка или предоставление определённому лицу или группе лиц прав на выполнение определенных действий.

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

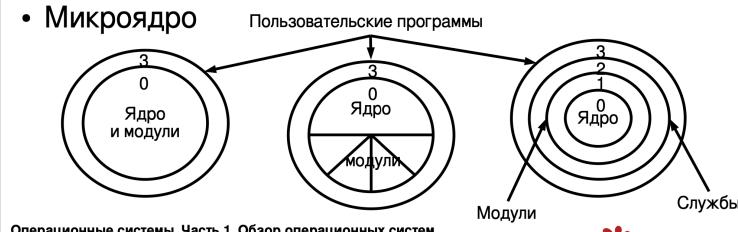
ЯДРО ОС



Есть пользовательский режим, есть режим ядра, есть интерфейс вызовов, где данные передаются между пользовательскими процессами и самой системой.

4 подсистемы: управления памятью, управления процессами и потоками, файловая и сетевая. Они работают на основе драйверов (частей кода, взаимодействующих с устройствами в составе компьютера) и платформо-зависимого кода (частей ядра, зависящих от архитектуры).

- Монолитное ядро
- Ядро с динамически загружаемыми модулями
- Микроядро



1. **Монолитное ядро** - есть служебная программа, информация загружается в память и там работает; ядро нельзя изменить без перекомпиляции.

Сейчас в чистом виде не используется нигде, кроме встроенных систем.

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

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 они 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: insmod, modprobe, rmmod. Удобно включить/выключить, не перегружая всю машину.

3. Микроядро

Микроядро (microkernel)

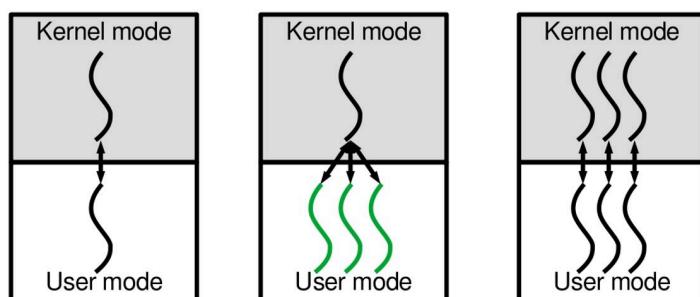
Небольшое привилегированное ядро операционной системы, обеспечивающее планирование процессов, управление памятью и службы связи и опирающееся на другие процессы для выполнения некоторых функций, традиционно связываемых с ядром операционной системы

Ядро выполняет базовые функции (диспетчеризация процессов, выделение памяти), а остальные функции осуществляются в виде сервисов, размещаемых на разных уровнях безопасности. Работает медленно за счет частого переключения контекстов. (пример: Windows NT)

10. Потоки исполнения, многопоточность, модели многопоточности.



- Поток (нить выполнения, thread) — единица диспетчеризации и выполнения ОС
- Posix Threads



Многопоточность - технология, при которой процесс, выполняющий приложение, разделяется на несколько одновременно выполняемых потоков.

Поток (thread) - диспетчеризуемая единица работы, включающая контекст процессора (в который входит содержимое счетчика команд и указателя вершины стека), а также собственную область стека (для организации вызова подпрограмм). Может быть прерван при переключении процессора на обработку другого потока.

Процесс - набор из одного или нескольких потоков, а также связанных с этими потоками системных ресурсов (таких, как область памяти, в которую входят код и данные, открытые файлы, различные устройства). Эта концепция очень близка концепции выполняющейся программы. Разбивая приложение на несколько потоков, программист получает все преимущества модульности приложения и возможность управления связанными с приложением временными событиями.

Многопоточность полезна для приложений, выполняющих несколько независимых задач, которые не требуют последовательного выполнения. Например: сервер, обрабатывающий запросы клиентов.

Создание процесса требует много ресурсов -> Слишком медленно!

Пример: на Apache 1.0 под каждый запрос порождался новый процесс, из-за чего он выполнялся долго, а CPU сильно загружался, в результате чего сервер можно было быстро положить. Поэтому было предложено создание новых потоков под каждый запрос в рамках одного процесса.

Стандартная модель - один поток на процесс.

Существует библиотека порождения потоков в Unix, использующаяся также в Windows (для совместимости) и разработке на Си - POSIX Threads.

В программировании **зеленые потоки (green threads)** — это потоки выполнения, управление которыми вместо ОС 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.

Из АК (надо удивить клима):

- потоки могут диспетчеризоваться внутри виртуальной машины (Erlang), в которой отслеживаются счетчики и когда надо, то переключаемся
- через “вставки” во время компиляции Go

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

Солярис умер в Java 1.2 когда переходили с зеленки на нормальные потоки.

11. Симметричная и асимметричная многопроцессорная обработка.



SMP vs ASMP

- 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 \neq Многопроцессорность

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



ASMP - некоторые процессоры “равнее других” (например, CPU готовит данные, подчинённые GPU обрабатывают).

“Простота” в кавычках, т.к. из-за одновременной работы на нескольких процессорах возможны блокировки.

Если процессор выйдет из строя, его работу могут взять на себя остальные -> надежно!

При увеличении кол-ва процессоров повышается производительность -> горизонтальное масштабирование!

Можно реализовать динамическое добавление ресурсов.

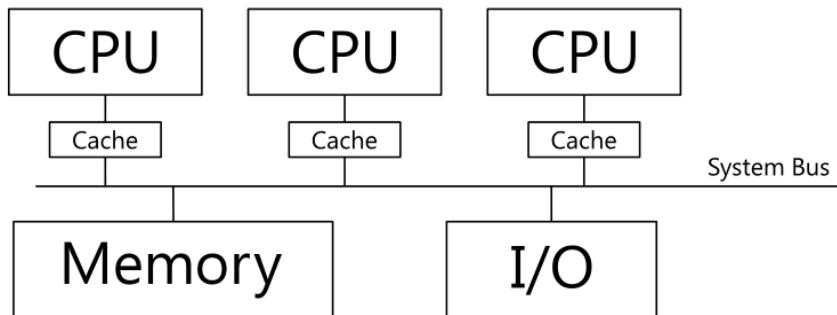
Многопоточность не синонимична многопроцессорности, потому что можно на одном процессоре запустить несколько потоков, а можно на системе с большим кол-вом процессоров однопоточный процесс, но юзаться будет только один.

Побольше инфы и воды:

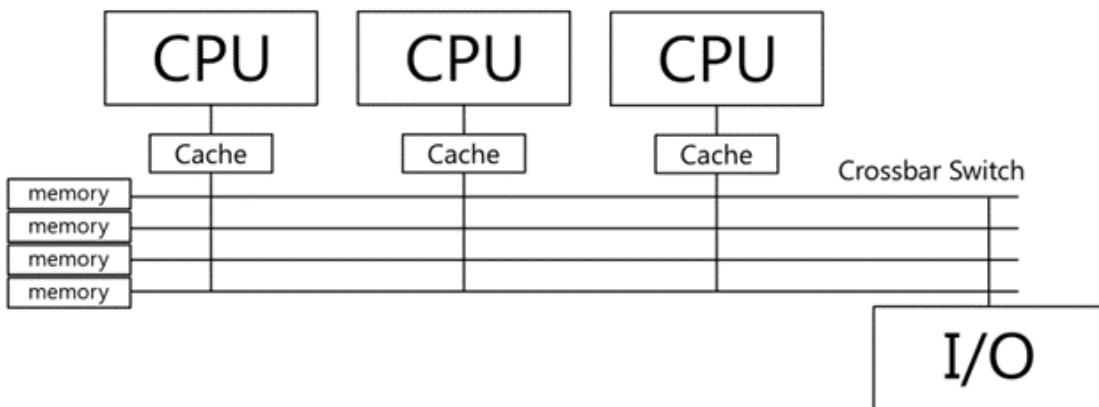
Симметричная многопроцессорность — архитектура многопроцессорных компьютеров, в которой два или более одинаковых процессора сравнимой производительности подключаются единообразно к общей памяти (и периферийным устройствам) и выполняют одни и те же функции. Каждый процесс управляет собой (диспетчеризует себя).

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.



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.



12. Виртуализация. Типы виртуализации.

Виртуализация — предоставление набора вычислительных ресурсов **(для выполнения задач)**, абстрагированное от аппаратной реализации. Обеспечивается логическая изоляция друг от друга вычислительных процессов, выполняемых на одном физическом ресурсе.

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.

Интерпретаторы работают под управлением основной машины и исполняют машинно-независимый код, т.е. поднимают уровень абстракции и позволяют реализовать то, что в машинных кодах сделать нельзя.

Контейнеры приложений — выделение дополнительного пространства пользователя; 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 (**software** или **hardware** - запускаем другую ОС, в первом случае эмулируется платформа, в последнем используется; соответственно, нельзя запустить, например, PS3 на линуке через **хардварь**), 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. При выходе машины из строя данные с неё мигрируют на другую машину. Provisioning - выделение пользователю облачных ресурсов и сервисов.

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



Отказоустойчивость

- 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



Отказы (faults) в системах

- Ошибочное состояние аппаратуры или ПО в результате сбоя компонентов
- Ошибки оператора (**мем не исполнять!**)
`perl -e '$??s::s:s:$?:s;:=]=>%-{<-|}<&|`{;;y; -/-:@[-`{-} ;`-{"/` -;;s;$_;see'`
- Физические помехи окружающей среды
- Ошибки проектирования, программирования, структур данных и пр.
- Могут быть: постоянные, временные (однократные или периодические)

Отказоустойчивость - это способность системы продолжать работу при аппаратных или программных ошибках.

Отказ или **сбой** - наблюдаемое проявление дефекта, в том числе падение программы. Может привести к невозможности выполнить задачу, получить верный результат.

Причины отказов:

- Ошибочное состояние аппаратуры или ПО в результате сбоя компонентов

- Ошибки оператора (ввел что-то не то)
- Физические помехи окружающей среды (кабель влияет на кабель)
- Ошибки проектирования, программирования, структур данных и пр.

Типы отказов:

- Однократные - например, изменение или сбой при передаче бита из-за импульсного шума или внешнего излучения
- Периодические - в разные, непредсказуемые моменты времени (например, когда неплотное соединение приводит к кратким потерям связи)
- Постоянные

Способы борьбы:

- Избыточность аппаратуры (двойное, тройное резервирование) - вместо одного устройства используется несколько.
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 со всех 3х и берётся "большинство" голосов.
- Аппаратная "горячая" замена компонентов (диски, контроллеры, процессоры, системные платы)
- Программная поддержка ОС выведения компонентов из системы и их подключения (нужно учитывать, если на компоненте лежит ядро)
- Организация уровней хранения RAID (Redundant Array of Inexpensive Disks) в дисковой подсистеме (смотреть [тут](#)).

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

Надежность (Reliability)

- $R(t)$ - Вероятность бесперебойной работы системы до времени t , при условии ее корректной работы в $t=0$
- Бесперебойная работа — корректная работа и защита данных
- Среднее время наработки на отказ (Mean Time To Failure)
$$MTTF = \int_0^x R(t) dt$$

Операционные системы. Часть 1.
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

Среднее время восстановления (Mean Time To Recover)

- Обычно время для перезагрузки, ремонта или замены неисправного компонента, установки (или переустановки) ОС и ПО

График времени восстановления (MTTR) показывает рабочее время (uptime) и время неработоспособности (downtime) в течение времени t . Время неработоспособности разбивается на три периода: Boot time, Reboot time и Repair time. Рабочее время (uptime) обозначено как U_1 , U_2 и U_3 .

$$MTTF = \frac{U_1 + U_2 + U_3}{3}$$
$$MTTR = \frac{\text{Boot time} + \text{Reboot time} + \text{Repair time}}{3}$$

Операционные системы. Часть 1.
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

Коэффициент доступности (Availability)

- Доля времени (%), когда система или служба доступна для запросов пользователей
- Простой (downtime) — время, в течении которого система недоступна
- Безотказная работа (uptime) — время, когда она находится в продуктивной работе

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR}$$

Надежность - свойство системы сохранять во времени способность выполнять требуемые функции в заданных режимах и условиях применения.

MTTF (mean time to failure) - среднее время наработки на отказ

MTTR - среднее время восстановления

Доступность - вероятность, что система находится в рабочем состоянии

Времяя простоя (downtime) — интервал с момента неработоспособности сервиса до момента возобновления его работы.



Классы доступности систем

Класс	Коэф. доступности	Время простоя в год
Непрерывная работа	1,0	0
Высокоотказоустойчивый	0,999999	32 секунды
Отказоустойчивый	0,99999	5 минут
Восстанавливаемый	0,9999	53 минуты
Высокодоступный	0,999	8,3 часа
Обычный	0,99-0,995	44-87 часов

Выделяется условно (не прижились классы).

Обычный - обычный ПК

Отказоустойчивый - кластерные системы. Не обязаны работать все ноды кластера.

Например, есть кластер из 2х нод, они могут работать по очереди (и пока один работает, другой может находиться в ремонте), главное, чтобы запросы пользователя обрабатывались.

Высокоотказоустойчивый - например, есть 2 вирт. машины: главная и подчинённая. При работе с главной ВМ, все изменения в памяти копируются на вторую ВМ. Таким образом, при выходе из строя главной ВМ, система очень быстро переключается на вторую ВМ

Непрерывная работа - практически недостижимо в реальности

15. Резервирование и отказоустойчивость.

Отказоустойчивость - способность системы продолжать работу при аппаратных или программных ошибках.

Причины отказов:

- Ошибочное состояние аппаратуры или ПО в результате сбоя компонентов
- Ошибки оператора (ввел что-то не то)
- Физические помехи окружающей среды (кабель влияет на кабель)
- Ошибки проектирования, программирования, структур данных и пр.

Типы отказов:

- Однократные сбои - например, изменение или сбой при передаче бита из-за импульсного шума или внешнего излучения
- Периодические сбои - в разные, непредсказуемые моменты времени (например, когда неплотное соединение приводит к кратким потерям связи)

Способы борьбы:

- избыточность аппаратуры (двойное, тройное резервирование) - вместо одного устройства используется несколько.
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 со всех 3х и берётся “большинство” голосов.

- аппаратная “горячая” замена компонентов (диски, контроллеры, процессоры, системные платы)
- программная поддержка ОС выведения компонентов из системы и их подключения (нужно учитывать, если на компоненте лежит ядро)
- организация уровней хранения RAID (Redundant Array of Inexpensive Disks) в дисковой подсистеме (смотреть [тут](#)).

Отказоустойчивость при ошибках аппаратуры предполагает избыточность (резервирование)

Методы резервирования:

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

В общем случае отказоустойчивость системы обеспечивается путем внесения в нее избыточности. К методам резервирования относятся следующие.

- **Пространственная (физическая) избыточность.** Физическая избыточность предполагает использование нескольких компонентов, которые одновременно выполняют одну и ту же функцию или настроены так, что один компонент доступен в качестве резервной копии, включающейся в случае сбоя другого компонента. Примерами могут служить использование нескольких параллельных схем или резервный сервер доменных имен в Интернете.
- **Временная избыточность.** Временная избыточность включает повторение выполнения функции или операции при обнаружении ошибки. Этот подход эффективен для временных ошибок, но непригоден для постоянных сбоев. Примером является ретрансляция блока данных при обнаружении ошибки передачи, как это делают протоколы управления линиями передачи данных.
- **Информационная избыточность.** Данная избыточность обеспечивает отказоустойчивость путем репликации или кодирования данных таким образом, чтобы ошибки в отдельных битах могли быть обнаружены и исправлены. Примером являются схемы с коррекцией ошибок, используемые в системах памяти, а также методы коррекции ошибок в RAID-дисках, о чем будет рассказано позже.

Также отказоустойчивость предполагает возможность замены компонентов системы без остановки ее работы

Методы повышения отказоустойчивости ОС

- Изоляция процессов (процессы внутри пользовательского АП изолированы друг от друга).
- Разрешение блокировок при параллелизме.

- Виртуализация - полностью изолировать выполнение (гипервизор с гостевыми ОС, при ошибке в одной из них рухнет только она)
- Точки восстановления и откаты (создаются и сохраняются копии ключевым файлов, позволяя в случае ошибки откатиться в нормальное состояние)

Ряд методов поддержки отказоустойчивости могут быть включены в программное обеспечение операционных систем. Ряд примеров будет встречаться вам на протяжении всей книги. В следующем списке приведены некоторые из примеров.

- **Изоляция процессов:** как упоминалось ранее в этой главе, процессы обычно изолированы один от другого с точки зрения основной памяти, доступа к файлам и потока выполнения. Структура, предоставляемая операционной системой для управления процессами, обеспечивает определенный уровень защиты от сбояного процесса других процессов.
- **Управление параллелизмом:** в главах 5, “Параллельные вычисления: взаимоисключения и многозадачность”, и 6, “Параллельные вычисления: взаимоблокировка и голодание”, будут обсуждаться некоторые трудности и ошибки, которые могут возникнуть при взаимодействии процессов или обмене информацией между ними. В этих главах будут также рассмотрены методы, используемые для обеспечения корректной работы и восстановления после сбоев, таких как, например, взаимоблокировка.
- **Виртуальные машины:** виртуальные машины, которые будут рассмотрены в главе 14, “Виртуальные машины”, обеспечивают более высокую степень изоляции приложений, а следовательно, и изоляцию сбоев. Виртуальные машины могут также использоваться для обеспечения избыточности, когда одна виртуальная машина выступает в качестве резервной копии для другой.
- **Точки восстановления и откаты:** точка восстановления представляет собой копию состояния приложения, сохраненную в некотором устройстве хранения, защищенном от рассматриваемых сбоев. Откат перезапускает выполнение из ранее сохраненной точки восстановления. При возникновении сбоя состояние приложения откатывается до предыдущей точки восстановления и перезапускается. Этот метод может использоваться для восстановления как после временных, так и после постоянных аппаратных сбоев и определенных типов сбоев программного обеспечения. Системы управления базами данных и транзакциями обычно обладают такими возможностями, встроенными в сами системы.

Повышение отказоустойчивости производится для повышения надёжности системы. Как правило, увеличение отказоустойчивости (и соответственно, повышение надежности) имеет определенную стоимость, либо финансовую, либо выражющуюся в падении производительности (либо и то, и другое одновременно). Таким образом, определение желаемой степени отказоустойчивости должно учитывать, что именно является критическим ресурсом.

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



«Отцы-основатели»

- MIT
 - Compatible Time-Sharing System (CTSS), IBM 709, 1961
 - Incompatible Timesharing System (ITS), PDP-10, 1967
- Манчестерский университет: Супервизор Atlas и экстракоды, Atlas, 1962
- MIT, GE, Bell Labs: Multics, 1965
- IBM: OS/360: Мейнфреймы System/360, 1966
- Technische Hogeschool Eindhoven: THE, Electrologica X8, 1968

Первые ОС.

- 1) CTSS - заложила функции, характерные для современных ОС. (MIT)
- 2) ITS - тоже разработана в MIT
- 3) Atlas - "в экстракодах реализовано многое из того, что вы уже знаете и сегодня активно используется".
- 4) Multics - прародитель UNIX.
- 5) OS/360 - даже сейчас работает на соотв. мейнфреймах, т.к. они отличаются высокой надёжностью, но имеют специф. ЯП и требуют дорогого обслуживания.
- 6) Technische Hogeschool Eindhoven



- UNICS: Bell Labs, 1969, Кен Томпсон, Деннис Ритчб и Брайан Керниган
 - Ключевые понятия: вычислительный процесс и файл
 - Компонентная архитектура: принцип «одна программа — одна функция»
 - Минимизация ядра
 - Независимость от аппаратной архитектуры и реализация на С
 - Унификация файлов

UNIX(CS) - потомок MULTICS, названный в пику ему.

Файл - лучшая абстракция для подсчёта и организации ресурсов, “в UNIX все есть файл”. Для рабочих программ и запускаемых задач - понятие вычислительного процесса.

В первых UNIX вместо мультитулов (типа perf) раскладывали сложные задачи на маленькие и создавали большое кол-во мелких утилит (одна функция на программу). Если хорошо их изучить, то можно объединять в конвейеры и скрипты для решения сложных задач, что иногда может быть быстрее других способов.

Минимизация ядра - четко перечисленный узкий функционал ядра, предотвращение неконтролируемого усложнения; остальные функции вынесены на уровень пользователя (например, оконная система), ядро проще поддерживать, разрабатывать остальное на стороне юзера эффективнее.

Независимость от аппаратной архитектуры и реализация на Си → Переносимость (но раньше считалось крамольным, так как память была малой, машины маломощными, а код ВУЯП надо переводить в бинари)

Унификация файлов (всё есть файл?)



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

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Unix SUS - набор спецификаций для совместимости продуктов разных разработчиков под Unix. Компании TOG и AG определяют, является ли система Unix.

TOG появилась после слияния в 1996 г. X/Open и Open Software Foundation (включил, т.к. первая будет упоминаться).

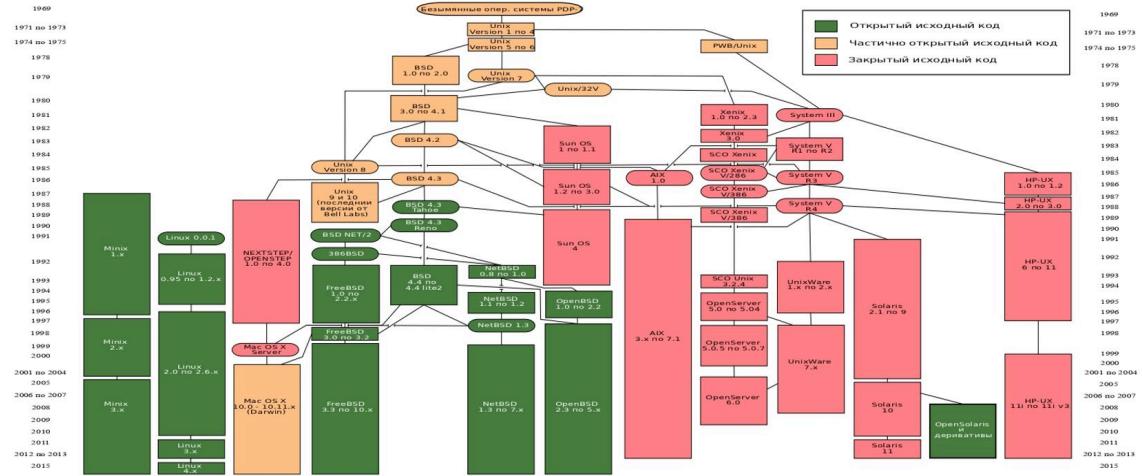
Сертификационный документ включает в себя ряд обязательных категорий (из Вики):

- **Основные определения** (*Base definitions*) — список основных определений и соглашений, используемых в спецификациях, и список заголовочных файлов языка Си, которые должны быть предоставлены соответствующей стандарту системой.
- **Системные интерфейсы** (*System interfaces*) — список системных вызовов языка Си.
- **Оболочка и утилиты** (*Shell and utilities*) — описание утилит и командной оболочки sh, стандарты регулярных выражений.
- **Пояснения** (*Rationale*) — объяснение принципов, используемых в стандарте.
- **X/Open Curses** — реализация (авторства X/Open) библиотеки управления терминалом Unix-подобных ОС для создания приложений с текстовым интерфейсом пользователя (также авторы X Window System).

На сегодня ведущими из Unix являются потомки System V, такие как наиболее распространённые на рынке AIX, HP-UX и Solaris, но в последнее время они теряют позиции.

Unix-like - имеют другое ядро и внутренние интерфейсы, но используют интерфейс POSIX, описывающий взаимодействие между ядром ОС и польз. приложениями.

Генеалогическое дерево



<https://commons.wikimedia.org/w/index.php?curid=48637753>

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Из Multics сформировался Unix, с 5-6 версий разделился на BSD и System V.

BSD: MacOS, FreeBSD, NetBSD, OpenBSD, ...; последние юзаются редко, в основном в фильтрах и роутерах, но MacOS широко развилась благодаря графическим возможностям и выходу iPhone.

System V: Solaris.

Linux и Minix - это не Unix, а отдельные ветки (Unix-like - реализуют POSIX). Linux написан на основе Minix, но имеет новое ядро.

GNU/Linux

- Линус Бенедикт Торвальдс, 1991 г., ну вы знаете, GNU GPL, ядро, общее руководство
- Ричард Столлман, Свободное Программное Обеспечение с 1983 (GNU is Not Unix) — библиотеки и связка — 1992

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Линус Торвальдс в 1991 году на основании лицензии GNU GPL создал ядро Linux, сейчас продолжает его контролировать и занимается общим руководством.

Библиотеки и обвязка GNU (системные утилиты) были разработаны Ричардом Столлманом (идеологом свободного ПО) в 1992 году (e.g., glibc).

Можно сказать, что это две части Linux.

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 GNU² 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 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 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.

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.

17. Понятие дистрибутива, дистрибутивы Linux.



«Дистрибутив»

- Ядро
- Окружение
- Менеджер пакетов и обновлений
- Графическая подсистема
- Прикладные программы
- Поддержка



Дистрибутивы

- Коммерческие и «Community»
 - Ubuntu/Canonical
 - Red Hat Enterprise
 - SUSE
 - Oracle Enterprise
 - Astra Linux
 - Debian/Ubuntu
 - Fedora/CentOS
 - Open SUSE
 - ArchLinux
 - Gentoo

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

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Дистрибутив - это комплекс файлов и программных компонентов, собранных и сконфигурированных так, что их по сути можно использовать “из коробки”.

Все Unix-like системы в современном мире состоят из дистрибутивов, в отличие, например, от монолитной Винды. Люди решили, что набор пакетов для любой конкретной задачи или аудитории будет лучше, и начали их группировать.

Дистрибутив Linux - это ядро Linux + набор софта для прикладных задач. Ядро всегда берётся от Торвальдса и является общим для всех дистрибутивов. Набор софта же для каждого свой. Формально дистрибутив фиксирует версии ядра и прикладных программ и представляет из себя систему пакетов. Больше разницы нет - программы все одинаковые, берутся из одних репозиториев.

В набор софта входит “системный” софт (пакетный менеджер, файловый менеджер, графическая оболочка) и набор прикладных программ (текстовый редактор, текстовый процессор, браузер, аудио/видео-проигрыватели и т.п.) (но не обязательно - на Gentoo даже нужно ядро при установке компилировать).

Окружение - необходимый набор спец. библиотек и утилит.

Граф. подсистема - X, Wayland, ...

Пакетный менеджер - apt (Debian), pacman (Arch), rpm (Red Hat), ...

Часть дистрибутивов поддерживается сообществом, которое использует их и заинтересовано в их развитии (Arch, Debian, Gentoo), они являются опенсорскими и бесплатными. Другие создаются компаниями; они уже могут быть проприетарными (Goobuntu); опенсорскими, но платными (Red Hat Enterprise); или бесплатными (Ubuntu), но с платной поддержкой (траблшутинг).

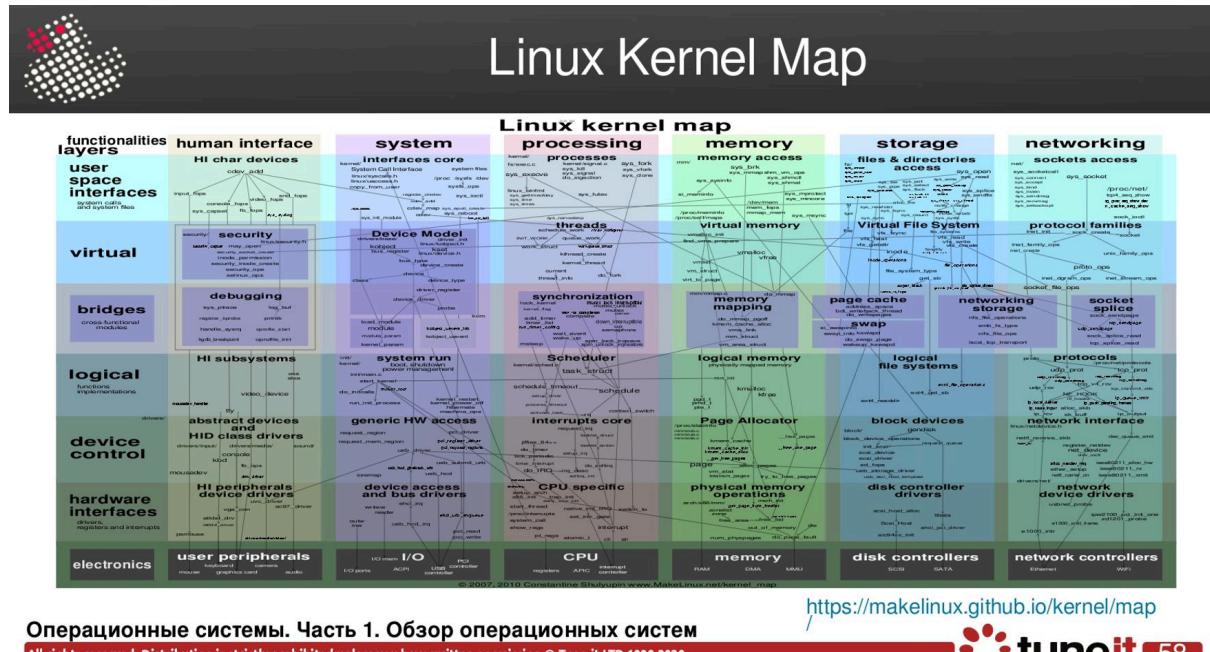
У дистрибутивов может быть разное назначение - от встроенных систем (OpenWrt) и ПК (Linux Mint) до суперкомпьютеров (Rocks Cluster Distribution). Есть дистрибутивы для специализированных решений. Например, Kali и Black Arch - инфобез, Astra Linux - военка, Ubuntu Studio - фото-, аудио- и видеостудия. Набор софта в них соответствующий.

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 утилиты GNU, поэтому обычно не причисляется к дистрибутивам Linux.

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

Билет не расписан, пускай пока будет так.



Приведённая Kernel Map достаточно логично разделена на отдельные функции и подсистемы ядра.

Здесь можно увидеть отдельную подсистему взаимодействия с пользователем, “системную часть системы”, которая обеспечивает связь системы, в том числе, например, “сископы”, которые осуществляют вызовы пользовательского пространства в саму систему; общие модели представления устройств внутри ОС; описание процесса запуска системы и так далее.

С точки зрения процессов на схеме представлено взаимодействие между процессами и трэдами на уровне блокировок, мьютексов, семафоров и т.д.

Приведенную схему удобно рассматривать и самостоятельно разбираться, как все устроено.

Попытка конспекта с очной лекции:

Выше - ближе к юзеру, ниже к процу

Fork на уровне ядра превращается в sysfork

Мониторы для синхронизации

Планировщики в ядре

Хардварные интерфейсы - самые низкие, управление прерываниями, учёт особенностей процессора

Управление памятью, вводом-выводом, сетевой интерфейс.

Модель драйверов устройств.

Virtual и bridges

Простейшие терминалы - shell обеспечивает интерактивную оболочку взаимодействия с системой

Сейчас в основном bash

Пользоваться Линуком даже без граф. интерфейса довольно удобно

Какой-то inet коммандер (первое могло послышаться)

Очень советовал перед второй лабой потыкать эту карту



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.
- Файловая система.
 - Предоставляет глобальное иерархическое пространство имен для файлов, каталогов и других объектов, связанных с файлами и функциями файловой системы.
- Драйверы символьных устройств.
 - Управление устройствами, которые требуют от ядра отправки или получения данных по одному байту, например терминалами, принтерами или модемами.
- Драйверы блочных устройств.
 - Управление устройствами, которые читают и записывают данные блоками, как, например, различные виды вторичной памяти (магнитные диски, CD-ROM и т.п.).



Основные подсистемы Unix/Linux (2)

- Сетевые протоколы. TCP/IP.
 - Поддержка пользовательского интерфейса сокетов для набора протоколов
- 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.
- Прерывания.
 - Обработка прерываний от периферийных устройств.
- Сигналы и IPC
 - Управляет межпроцессным взаимодействием

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Рассмотрим основные подсистемы Linux/Unix.

Планировщик - это то, что управляет процессами. Его задача - создавать процессы, а потом ими управлять, путём переключений добиваясь "справедливого" распределения ресурсов процессора между ними. О подходах к определению "справедливости" распределения мы поговорим позднее.

Следующая подсистема занимается организацией виртуальной памяти. Эта подсистема выделяет память, создаёт странички, удаляет странички, занимается свопингом, пэйджингом.

Подсистема управления физической памятью очень похожа на предыдущую подсистему Linux, с той лишь разницей, что управление производится не виртуальной памятью, а физической. Причём обе подсистемы управления памятью очень тесно взаимодействуют друг с другом.

Файловая система. В Unix всё является файловой системой. И любая система, которая работает с файлами, пишется в виде драйвера к этой подсистеме.

С точки зрения физических файловых систем существует большое количество специальных драйверов, в написании которых сегодня разработчики соревнуются, пытаясь создать наилучшие.

Все существующие в Linux драйверы можно разбить на два типа: драйверы символьных устройств и драйверы блочных устройств (в описании они различаются литерами "с" и "б").

На самом деле, особой разницы между ними нет. Просто по сути драйверы блочных устройств осуществляют буферизацию. Символьные драйверы непосредственно передают данные на устройство, при этом буферизация осуществляется в буферном кэше, куда эти данные помещаются и где хранятся промежуточно.

Характерный пример символьных устройств - это различные терминалы. Пример блочных устройств - это диски, CD-ROM, флэшки и другие устройства, на которые запись производится не побайтно, а только некими блоками, которые предварительно собираются в кэше.

Следующей основной подсистемой Linux является **сетевая подсистема** со всеми её сетевыми протоколами, в число которых входят не только TCP/IP, но и многие другие, связанные с сетевым взаимодействием. Например, в их число входит достаточно старый, но широко распространённый в банковской сфере (как средство межбанковского общения) протокол X25.

Сетевая подсистема занимает значительную часть ядра Linux, и средств для анализа трафика сети в ней едва ли не больше, чем всех остальных средств ядра.

Драйверы сетевых устройств. С появлением гипервизоров все привычные драйверы физических устройств пришлось виртуализировать, что привело к очень большому числу драйверов сетевых устройств, существующих в Linux сегодня.

Отдельно в операционной системе обрабатываются **ловушки и отказы**. Для их выявления у процессора есть множество различных прерываний, на которые нужно соответствующим образом реагировать, в том числе либо пуская систему “в панику”, либо подгружая дополнительные странички.

Подсистема обработки прерываний в основном обрабатывает прерывания от периферийных устройств.

Подсистема межпроцессного взаимодействия. Наиболее частый сигнал, обрабатываемый ею - это ликвидация процесса. IPC - стандартное средство межпроцессного взаимодействия, включающее в себя разделяемую память, семафоры, сообщения и т.п.

Перечень функциональности основных подсистем

Сетевые протоколы

Раньше TCP/IP не был глубоко интегрирован в ядро, и это было очень медленно

Одна сетевая карточка сильно жрёт ядро процессора, поэтому начали это дело оптимизировать

Сетевые протоколы сильно связаны с дровами сетевых устройств

Пакетный фильтр - какой-то wall

Паника сделана для того, чтобы собрать дамп ядра и понять, в чем ошибка

Записываться и весить дамп будет много, но потом можно поковыряться отладчиком и понять, аппаратная или программная ошибка

Корректоры памяти

Если видно, что ошибка все время постоянно в одном и том же месте то система, об этом напишет

Прерывания

Таймер обрабатывается на стороне ядра

Отдельная подсистема для общения между собой, есть сигналы типа sigkill, есть ipc как разделенная между процессами память (intimate shared memory), вместе с сигналами можно передавать некоторую информацию, можно передавать структуры

Сигналы это тоже медленно, гарантия доставки вызывает вопросы

В Oracle всё взаимодействие между процессами основывается на IPC

Список сигналов kill -l

IPC - стандартное средство межпроцессного взаимодействия.

Попытка конспекта с очной лекции:

Если запустить два процесса с помощью pmap, то они будут по адресам одинаковы

ВП предоставляет для процессов функции выделения памяти (malloc, mmap)

Именованную память можно удалить, потому что она всегда есть на диске

Анонимную нельзя, потому что она создана динамически и данные есть только в памяти, поэтому надо свопнуть

В ядре есть такое понятие как аллокатор. Ядро - это обычная программа, но туда, например, загружаются драйвера, и для них надо выделить память страницами. В ядре есть некоторое количество структур для описания, например, трендов или процессов в Линуке

SLAB-аллокатор (слабый локатор, kekekekekekekeke)

Виртуальный узел для файлов

В Unix всё есть файл, поэтому ФС система устроена достаточно гибко

/proc - виртуальная ФС

Разницы по большому счету между дровами символьных и блочных устройств нет

Символьные типа терминал посимвольный ввод-вывод

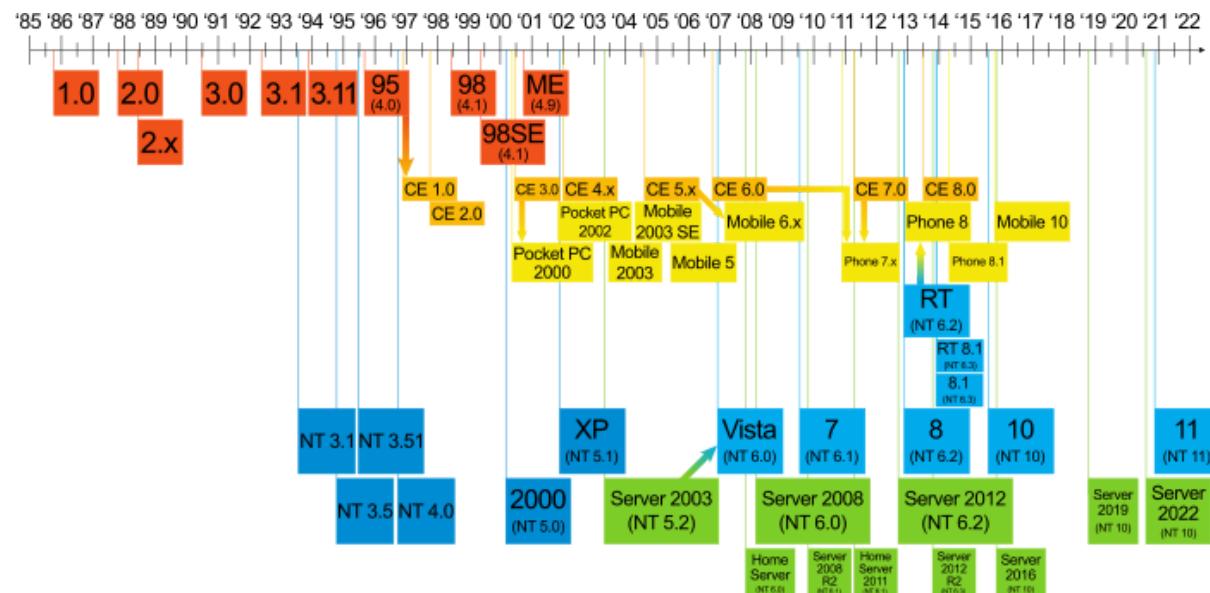
Символьные устройства = сырье

Блочные - кэшируют информацию в ядре

Блочный кэш в ядре, механизм для кэширования обеспечивается ядром

Крч, разница в промежуточном кэшировании

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 предназначенней 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.

Хронология развития Windows.

Версии 1.0 и 2.0 были по сути надстройкой над DOS и заменой Norton Commander для работы с файлами. В них применялась философия GUI MacOS.

Тогда в процессорах не было многих привычных теперь средств. Их память представляла собой плоское АП без виртуальной памяти, RAM = 640 Кбайт, использовалась сегментная адресация (в начале РОН была часть, отведенная под адрес). Процессоры 8086 были 16-разрядными, адрес был 20-разрядный. Позднее появилось виртуальное переключение страниц.

Первую значимую версию 3.11 ещё нельзя было назвать полнофункциональной ОС, т.к. там не была развита многозадачность.

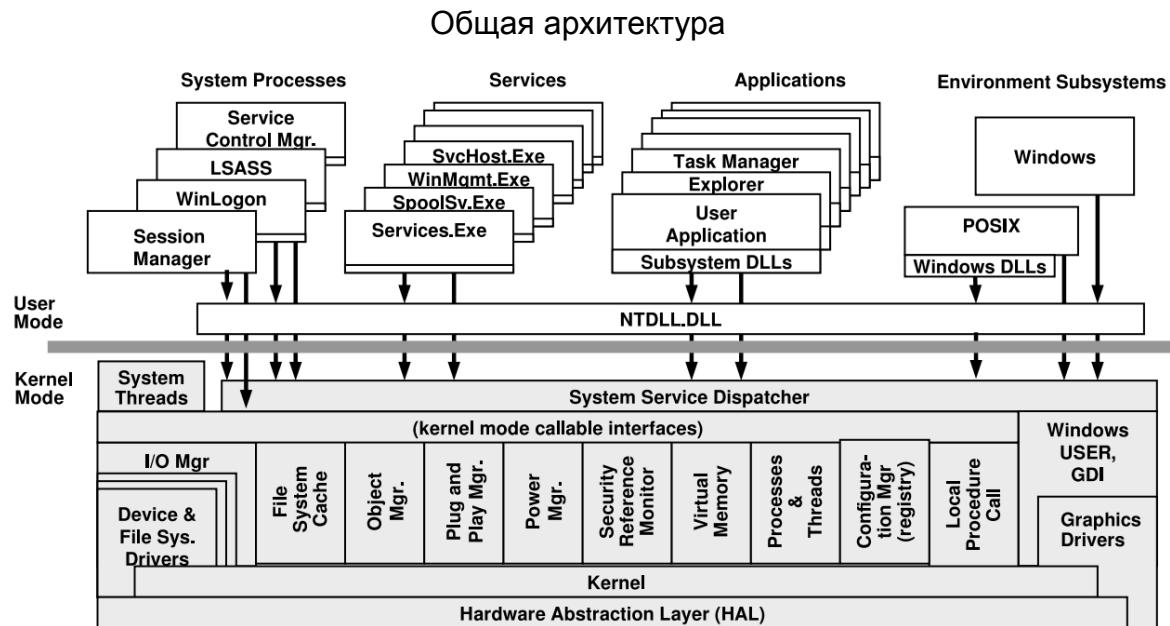
95 и 98 уже были “настоящими”, XP ещё живёт на старых и виртуальных машинах.

Последняя - 11 (2021!).

У Windows есть несколько “линеек”, в т.ч. мобильная и серверная.

CE (Compact) - бездисковые станции для тонких клиентов, иногда наработки используются в мобильных.

20. Общая архитектура Windows. Windows API



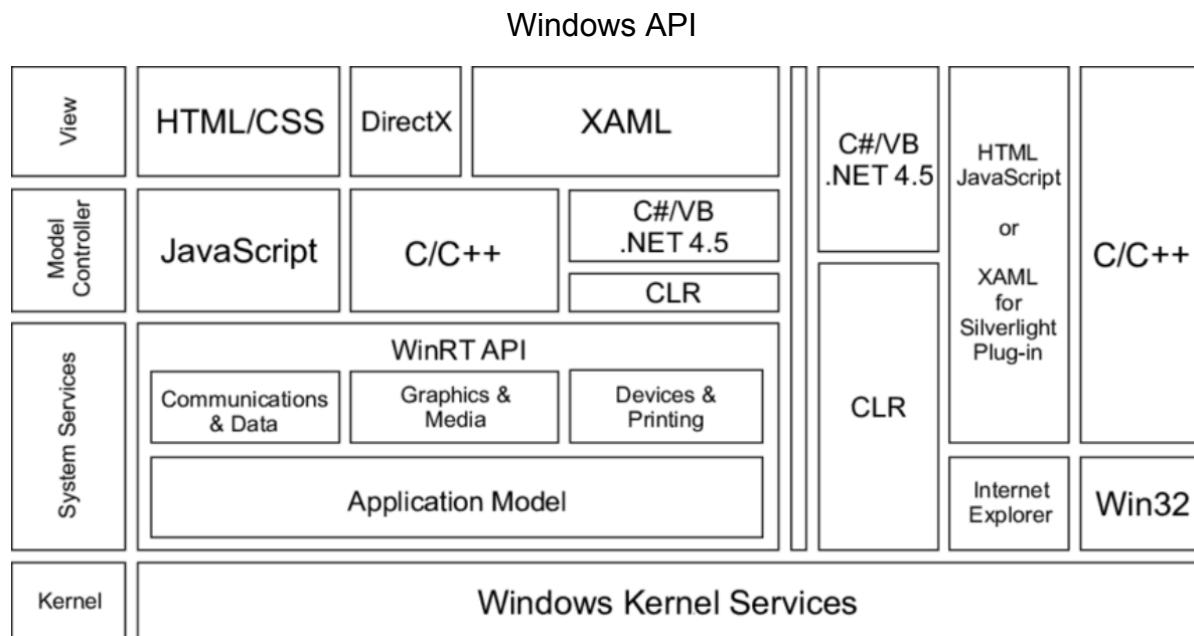
Пояснение:

С точки зрения архитектуры Windows похожа на UNIX. Есть процессы, которые могут быть как пользовательские, так и служебные, есть сервисы, приложения, окружения для работы. Есть интерфейс системных вызовов – ntdll.dll, эти вызовы работают в kernel mode. Схожая виртуальная память. Имеются подсистемы управления (энергией, виртуальной памятью, кешем, процессами и потоками, ...). Интерфейс взаимодействия с аппаратурой – HAL.

Services/System Processes - службы, которые стартуют когда пользователь входит в windows, и останавливаются и удаляются, когда выходит. Операционная система Windows содержит множество системных процессов, которые присутствуют каждый

раз, когда мы загружаем наши машины. Эти процессы отвечают за многое. От инициализации и создания пользовательского интерфейса до загрузки необходимых драйверов и DLL.

Environment Subsystems - окружения, которые эмулируют те или иные работы операционной системы. Например, эмулирующие POSIX, так мы можем запустить программу, написанную под “Linux” для работы под виндой.



Пояснение:

В винде есть общий подход к прикладным интерфейсам, с помощью которых мы можем разрабатывать собственные программы. Это Windows API. На фото мы видим стандартные. Системные сервисы представлены как WinRT, это довольно низкоуровневая вещь, так как она работает непосредственно с ядром. Есть различные API для языков программирования. Т.е. нам как разработчикам нужно изучить эту штуку, для того чтобы разрабатывать приложения на винде.

21. Сервисы, функции и важные компоненты Windows.

The slide is part of a presentation titled "Сервисы и функции" (Services and Functions). It lists various Windows API components:

- 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)
 - msrvct.dll, kernel32.dll

Operational systems. Part 1. Overview of operational systems
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-It LTD 1999-2020

tuneit 67

Windows API не похожи на UNIX API, так как создавались самостоятельно. Если есть необходимость делать «переносимые программы», то в этом случае необходимо работать в Windows «в стиле POSIX».

В Windows можно создать процесс и файл, используя функции CreateProcess и CreateFile.

Если необходимо сделать вызов ядра операционной системы, то для этого можно воспользоваться функцией NtCreateSystemProcess. Это и есть тот самый системный вызов, который проходит сквозь границу ядра.

В Windows также есть набор дополнительных функций для обращения к ядру. Они не такие важные, как «сискоЛы». Их примером может, например, служить команда по созданию области памяти (ExAllocatePoolWithTag).

В операционной системе Windows также существуют windows-сервисы, которые могут что-нибудь вернуть после запроса.

И так же существует достаточно большой набор динамических библиотек DLL, которые всегда можно использовать и каждая из которых выполняет какие-то свои функции.

(В конспекте NtCreateUserProcess)

Короче рассуждаем так - сначала идут три уровня управления компонентами системы:

- WinAPI - высокогородные функции для создания процесса/файла и т.д.;
- Системные вызовы - то, во что потом преобразуются вызовы WinAPI и придут в ядро;
- Функции ядра (самый низкий уровень) - то, во что преобразуются системные вызовы, например аллокация памяти.

Далее идут всякие подключаемые сервисы для обслуживания пользователя, всевозможные DLL'ки, расширяющие базовый функционал 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

В Windows стандартно есть собственный гипервизор, который называется Hyper-V. Он позволяет запускать гостевые операционные системы. Эта опция платная, и большинство пользователей, для того, чтобы в «винде» запустить что-то другое, пользуются разработками попроще, например, VirtualBox.

Существует целый ряд устройств, код для которых надо загружать при самом старте устройства. В частности, к этой группе оборудования относятся разного рода сетевые карточки. Для производства загрузки в таких случаях в Windows есть специальная подсистема для загрузки Firmware.

Еще одной важной компонентой Windows является терминальный сервер – Terminal Servers.

Изначально Windows не создавалась в качестве системы, в которой будет работать много пользователей, и она рассматривалась как операционная система, предназначенная для одного пользователя. Поэтому для обеспечения работы с серверами потребовалось создание специальных терминальных сервисов, которые обеспечивали такое взаимодействие через «тонкие клиенты» для того, чтобы иметь возможность открывать удалённые сессии на «большом сервере» Windows.

Отдельная тема Windows связана с объектами и безопасностью объектов данных. Все эти решения сделаны в виде объектов. И ими можно управлять как объектами.

Реестры. В реестре Windows хранится конфигурационная информация. Этой информацией обычно начинают пользоваться для устранения серьезных неполадок в системе.

Отдельная тема администрирования в Windows – это оснастки. Оснастка – это специализированная программа, которая администрирует определенный пул задач и представляет собой некий законченный интерфейс для конкретного набора административных действий. В Windows на сегодняшний день разработано большое количество оснасток. Местами они пересекаются по функционалу.

В линуксе тоже есть Firmware. Работает по такому же принципу - загружает микрокод.

22. Процесс, характеристики процесса в момент выполнения.

Состояние процесса. Разделение ресурсов.

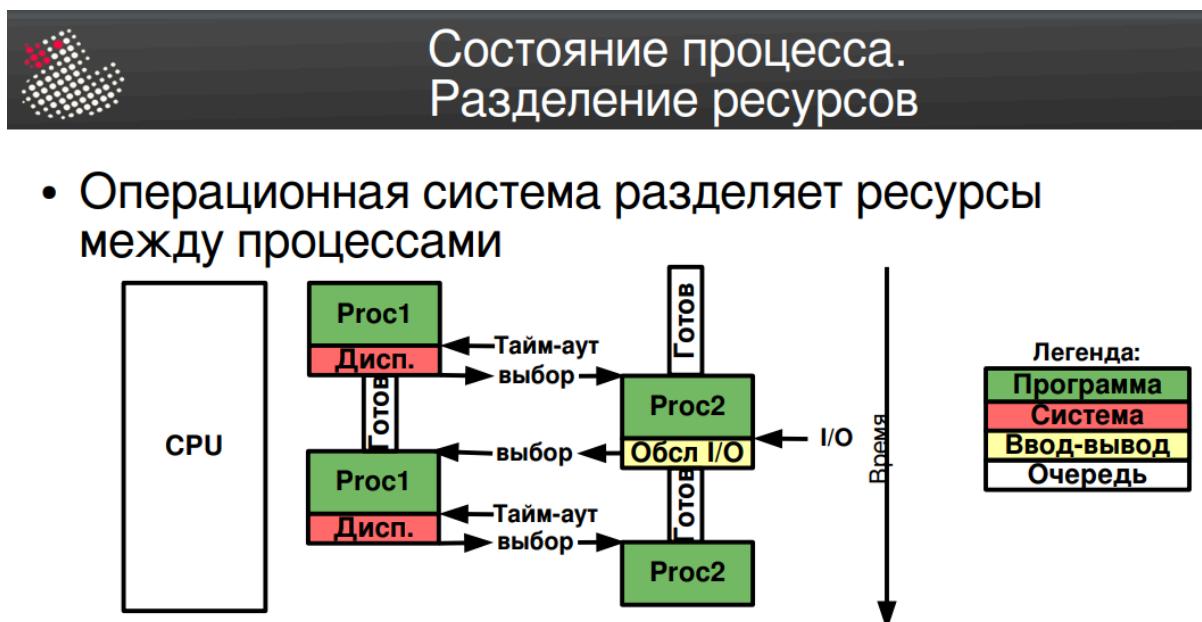


Итак, процесс это:

- Выполняемая программа
- Экземпляр программы, выполняющейся на компьютере
- Сущность, которая может быть назначена процессору и выполнена на нем
- Единица активности, характеризуемая выполнением последовательности команд, текущим состоянием и связанным с ней множеством системных 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

Модель процесса с двумя состояниями (выполняется, не выполняется) не является успешной. К примеру, используя Циклический метод(round robin), реализация такой модели не является адекватной. (Один процесс не выполняется, но готов. Другой процесс не выполняется и ждет поступления данных с ВУ). То есть используя модель с двумя состояниями, мы ищем не заблокированные процессы в очереди не выполняемых, а потом только их подгружаем. Логичным было бы разделить не выполняющиеся процессы на готовые к выполнению и заблокированные.

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/Ожидаящий⁵.** 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.

Paging / Swapping



Paging/Swapping

- 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 (пейджинг) — выгрузка (и загрузка) неиспользуемых страниц процесса на диск
- Swapping (свопинг) — выгрузка всего процесса, кроме критически важных для ядра структур управления

Операционные системы. Часть 2. Процессы и потоки

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



14

Основной памяти всегда мало, поэтому можно вытеснять из памяти редко использующиеся страницы, сохраняя их на диск и загружая обратно по мере необходимости, что повышает производительность системы и позволяет искусственно освободить память ОЗУ. То есть, освобождать память для активно работающих программ за счет “выпихивания” более блокированных.

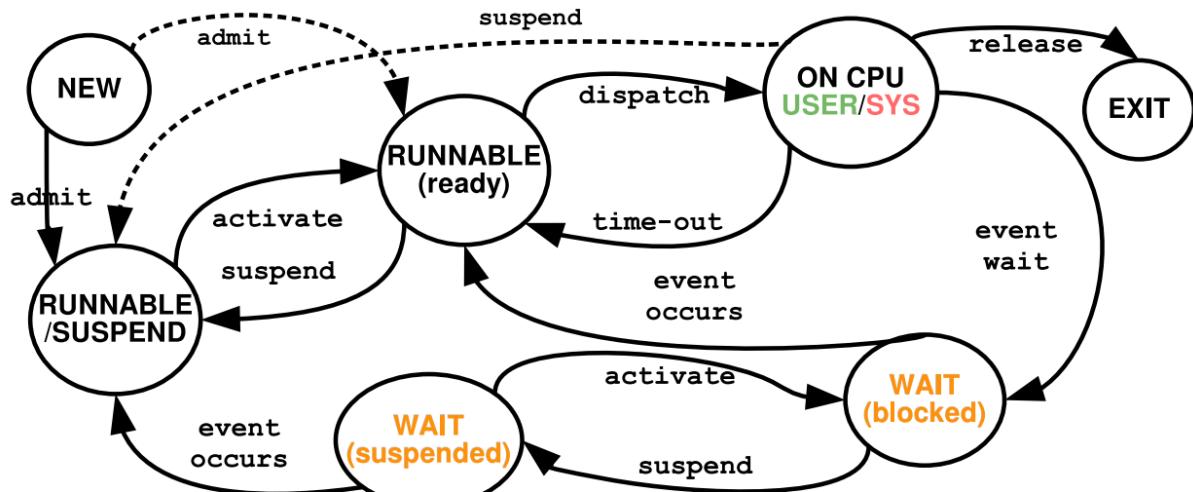
В Windows для этого используется файл pagefile.sys, а в Linux создаётся раздел swap.

Swapping (исторически первый) - выгрузка процесса на диск при неиспользовании целиком, а не отдельными структурами (что раньше было реально, т.к. процессы были маленькими, но сейчас надо много времени на выгрузку и загрузку обратно)

Потом появился Paging - выгрузка не всего процесса, а отдельных неиспользуемых страниц. То есть, если в течение заданного времени страница не используется, то считается неиспользуемой и помечается, а потом система в зависимости от содержания принимает решение - если код, то удаляет (всегда можно из файла загрузить заново, он в принципе неизменяемый), если данные, то сохраняет в swap, и т.д. Структуры ядра не выгружаются, потому что являются критически важными.



Модель процесса с 7-ю состояниями



Операционные системы. Часть 2. Процессы и потоки
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

На этой схеме всё те же старые состояния процесса (скопировано из 23его):

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/Ожидаящий⁵.** 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.

С пейджингом и свопом появляются новые состояния процесса.

1. **Runnable/Suspend** - процесс может выполниться, но весь или частично находится в свопе. Сюда можно попасть либо из New (операционная система хочет положиться на тот же алгоритм activate для вытаскивания нужных процессу страниц в память. Только в редких случаях процесс может быть быстро загружен в память, тогда он перескочит сразу в Runnable/Ready).
2. **Wait/Suspend** - в это состояние процесс переходит, когда ОС решает, что он слишком долго блокируется, или когда памяти мало. Система приостанавливает и выгружает его в своп. После прихода ожидаемого события ОС отправляет его в Runnable/Suspend.



Wait/Suspended state

- Процесс приостановлен и выгружен в область подкачки.
- Причины попадания в состояние:
 - Длительное ожидание событий операционной системы
 - Недостаток памяти (зачем держать в памяти процесс, который не имеет возможности исполняться)
- По событию «suspend» процесс выгружается на диск
- По событию «activate» загружается в основную память
- Повышенная нагрузка на дисковую подсистему!



Runnable/Suspended state

- Процесс готов к выполнению, но он выгружен из памяти
- Почему?
 - Был неготов к выполнению и выгружен но произошло событие, которое позволяет выполнится
 - «Desperate memory conditions»
 - Команда пользователя
 - Создание процесса в «минимальном» варианте, без, например, создания сегментов памяти



Причины приостановки процессов

- **Swapping**
 - ОС нужно освободить память, чтобы загрузить готовый к исполнению процесс
- **Другие причины ОС**
 - Приостановка фонового, служебного или «подозрительного» процесса
- **Интерактивный запрос пользователя**
- **Запрос родительского процесса**
- **Задание режима времени исполнения**
 - Периодических характер исполнения процесса

25. Управляющие таблицы процесса. Образ процесса.

4 основных подсистемы ядра и описывающие их внутренние структуры:

1. Память
2. Файлы
3. Устройства
4. Процессы

Внутри ОС существуют списки процессов (например, глобальный список процессов и список состояний, в которых они могут пребывать)

Каждый процесс имеет структуру образа, куда входят ядерная и пользовательская части (описывают и характеризуют процесс, а также содержат данные процесса и др.) В частности, существует подсистема с её собственными внутренними структурами; существуют файлы и структуры файловой системы, обеспечивающие работу с файлами. Аналогичными структурами подсистем можно описать устройства и процессы.

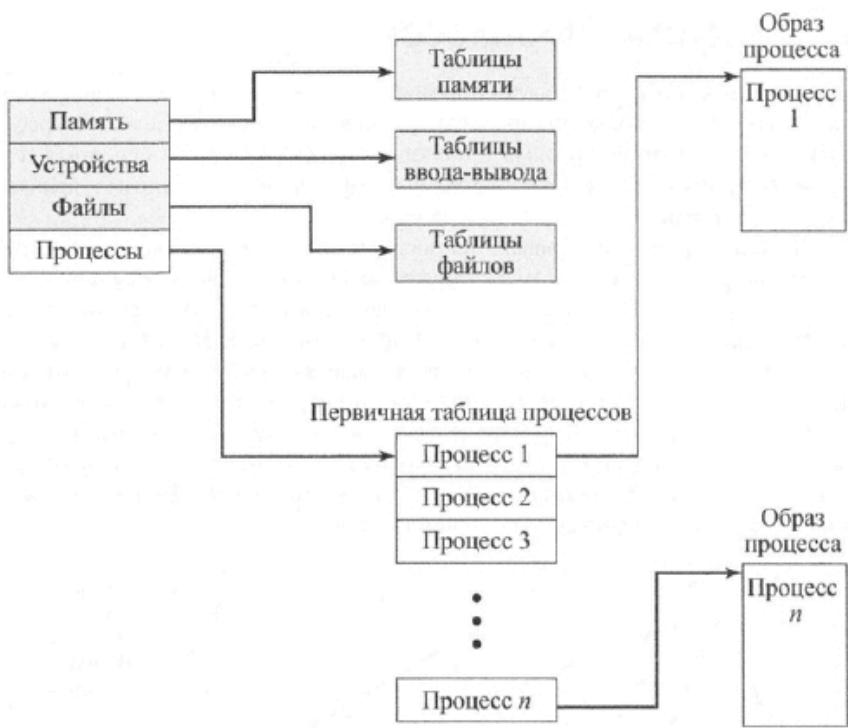
Внутри операционной системы существуют списки процессов. Примерами таких списков могут являться “глобальный список” процессов и “список состояний”, в которых процессы могут находиться.

Ядро хранит информацию о том, что происходит с процессом, в каком он состоянии, что ему нужно для исполнения, сколько времени потрачено на CPU и т.д.

По команде `rtar` можно получить карту памяти, по которой можно определить, где находятся код, данные, куча, стек и т.д.

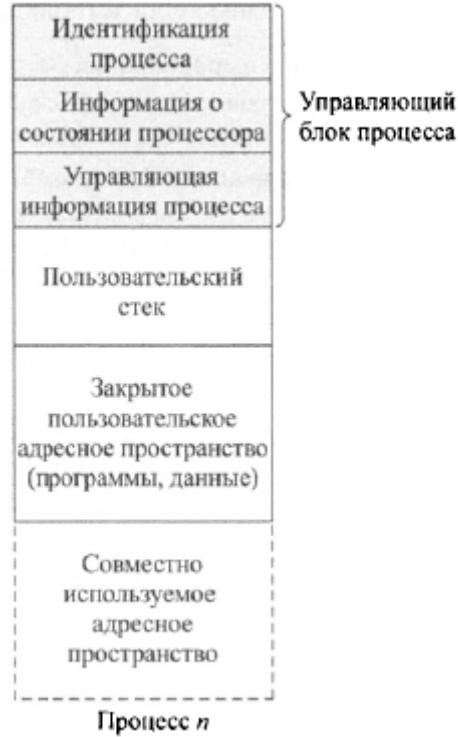
Kbytes - размер сегмента, RSS - размер страниц в ОЗУ, Dirty - кол-во измененных страниц, Mode - права, Mapping - соответствующий файл (также м.б. анонимная память или стек).

Статические переменные хранятся в сегменте данных, внутри функций - в стеке, обычные - в куче.



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 надо выполнить, неизменяя ее)
- **Системный стек** (хранение параметров, адресов вызова процедур и системных служб)
- **Управляющий блок процесса** (данные, необходимые ОС для управления процессом)



```
#include <stdio.h>
int var_int_data[1024];
static char var_char_data[4096];

void foo(int var_inc) {
    int var_a = 10;
    static int var_sa = 10;
    var_a += var_inc;
    var_sa += var_inc;
    printf("a = %d, sa = %d\n", var_a, var_sa);
}
int main(int argc, char**argv) {
    int var_i;
    for (var_i = 0; var_i < 10; ++var_i)
        foo(var_i);
}
```

Address	Kbytes	RSS	Dirty	Mode	Mapping
000055555554000	4	4	4	r----	a_prog
000055555555000	4	4	4	r-x--	a_prog
000055555556000	4	0	0	r----	a_prog
000055555557000	4	4	4	r----	a_prog
000055555558000	4	4	4	rw---	a_prog
000055555559000	8	0	0	rw---	[anon]
00007ffff7dc0000	148	148	0	r----	libc-2.31.so
00007ffff7de5000	1504	484	8	r-x--	libc-2.31.so
00007ffff7e5d000	296	64	0	r----	libc-2.31.so
00007ffff7ea7000	4	0	0	----	libc-2.31.so
00007ffff7ea8000	12	12	12	r----	libc-2.31.so
00007ffff7ab000	12	12	12	rw---	libc-2.31.so
00007ffff7fae000	24	20	20	rw---	[anon]
00007ffff7fc0000	12	0	0	r----	[anon]
00007ffff7fce000	4	4	4	r-x--	[anon]
00007ffff7fcf000	4	4	0	r----	ld-2.31.so
00007ffff7fd0000	140	140	24	r-x--	ld-2.31.so
00007ffff7fd3000	32	32	0	r----	ld-2.31.so
00007ffff7fcf000	4	4	4	r----	ld-2.31.so
00007ffff7fd0000	4	4	4	rw---	ld-2.31.so
00007ffff7fe0000	4	4	4	rw---	[anon]
00007fffffd0000	132	12	12	rw---	[stack]
fffffffff600000	4	0	0	--x--	[anon]
total kB	2368	960	120		

```
serge@ra:/tmp$ gcc -o a_prog -g a.c
serge@ra:/tmp$ gdb a_prog
GNU gdb (Ubuntu 9.1-0ubuntu1) 9.1
[...]
(gdb) break foo
Breakpoint 1 at 0x1149: file a.c, line 7
(gdb) run
Starting program: /tmp/a_prog
Breakpoint 1, foo (var_inc=32767) at a.c:7
7 {
(gdb) step 2
10     var_a += var_inc;
(gdb) print &var_char_data
[...] &var_int_data, &var_a, &var_sa, &var_inc
$4 = (char (*)[4096]) 0x55555555040 <var_char_data>
$3 = (int *)[1024] 0x555555559040 <var_int_data>
$5 = (int *) 0x7fffffffde5c #var_a
$6 = (int *) 0x555555558010 <var_sa>
$7 = (int *) 0x7fffffffde4c #var_inc
```

Address	Kbytes	RSS	Dirty	Mode	Mapping
00005555555000	4	4	4	r-x--	a_prog
000055555559000	8	0	0	rw---	[anon]
00007ffff7dc0000	148	148	0	r----	libc-2.31.so
00007ffff7de5000	1504	484	8	r-x--	libc-2.31.so
00007ffff7fab000	12	12	12	rw---	libc-2.31.so
00007ffff7fae000	24	20	20	rw---	[anon]
00007ffff7fc0000	12	0	0	r----	[anon]
00007ffff7fce000	4	4	4	r-x--	[anon]
00007ffff7fcf000	4	4	0	r----	ld-2.31.so
00007ffff7fd0000	140	140	24	r-x--	ld-2.31.so
00007ffff7fd3000	32	32	0	r----	ld-2.31.so
00007ffff7fcf000	4	4	4	r----	ld-2.31.so
00007ffff7fd0000	4	4	4	rw---	ld-2.31.so
00007ffff7fe0000	4	4	4	rw---	[anon]
00007fffffd0000	132	12	12	rw---	[stack]
fffffffff600000	4	0	0	--x--	[anon]
total kB	2368	960	120		

26. Управляющий блок процесса (PCB), состав PCB

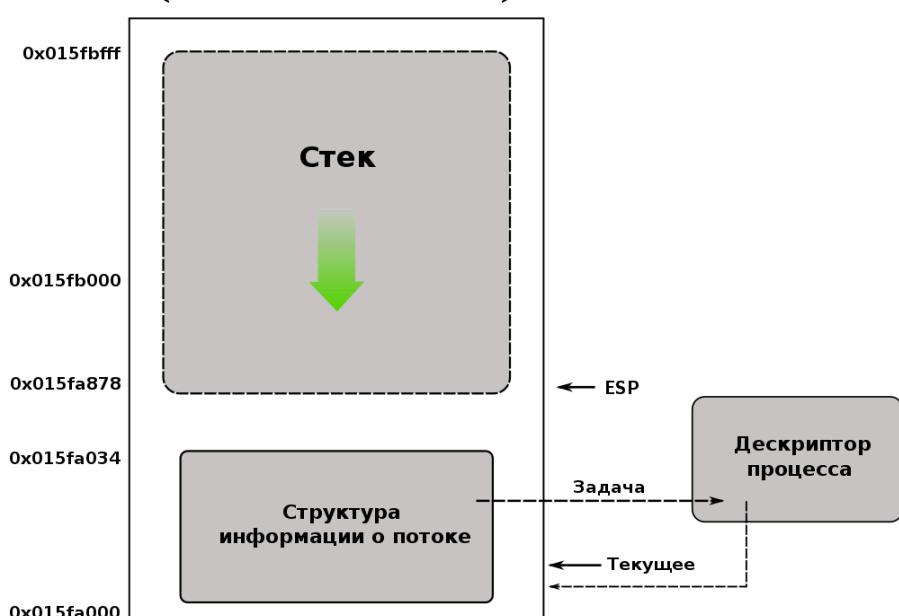


PCB (Process control block) - структура данных, используемая ОС для хранения всей информации о процессе (дескриптор процесса). Создается при инициализации процесса. Хранит следующую информацию:

- Идентификация процесса
- Информация о регистрах (необходима для переключения процессов)
- Другая информация, управляющая процессом (см. картинку)

В общем, можно сказать, что это самая важная структура данных в ОС, т.к. информация этих блоков читается и изменяется почти каждым модулем ОС (включая модули планирования, распределения ресурсов, обработки прерываний, контроля и анализа).

**Стек процесса ядра
(Process kernel stack)**



PCB должен храниться в памяти, защищенной от обычного доступа процесса. В некоторых ОС этот блок размещается в начале стека ядра процесса.

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



Функции ОС связанные с процессами

- Управление процессами
 - Создание и завершение процессов
 - Планирование и диспетчеризация процессов
 - Переключение процессов
 - Синхронизация и поддержка обмена информацией между процессами
 - Организация управляющих блоков процессов
- Управление памятью
 - Выделение адресного пространства процессам
 - Пейджинг и Свопинг
 - Управление страницами и сегментами
- Управление вводом-выводом
 - Управление буферами
 - Выделение процессам каналов и устройств ввода-вывода
- Функции поддержки
 - Обработка прерываний
 - Учет использования ресурсов
 - Текущий контроль системы



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

В Unix-like системах используется комбинация системных вызовов fork (копирование процесса), exec (замена образа процесса на новый). В Винде CreateProcess.



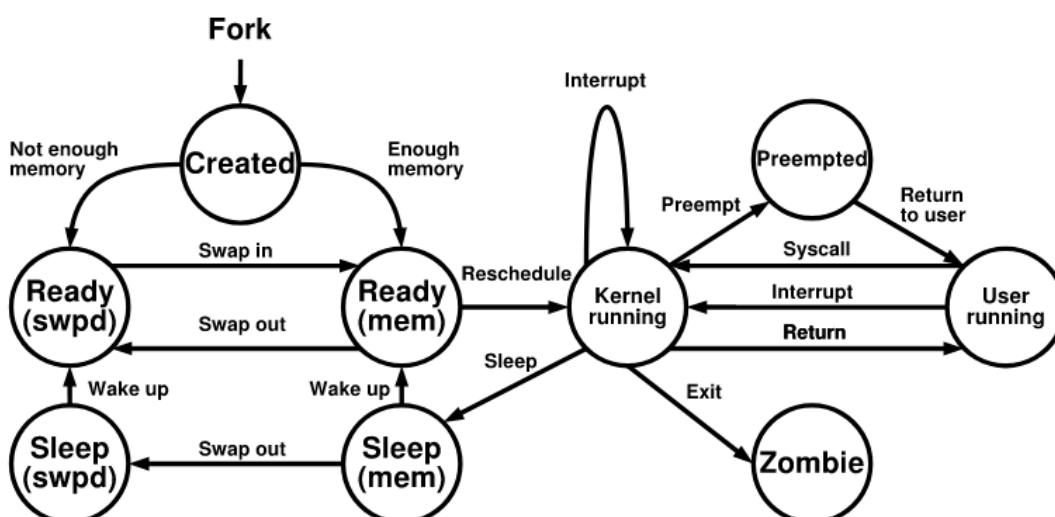
Переключение процессов

- 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 — отсутствие блока памяти

- При переходе в kernel mode необходимо сохранить информацию пользователя (например, РОН)
 - Для него используется механизм прерываний:
 - прерывания IO
 - обработка исключений
 - системные вызовы
- Пример - когда нам нужно открыть файл, генерируется с использованием сист. библиотеки соответствующий системный вызов, потом срабатывает ловушка и происходит переход в ядро, потом по таблице функций ищется нужный syscall и срабатывает функция, которая анализирует значения параметров и отсылает нас в соотв. часть ядра, после чего оно генерирует ответ.
- Ситуации перехода из user в kernel mode:
 - отработка прерывания таймера (конец кванта времени -> надо переключить процесс в Runnable и передать управление другому процессу)
 - прерывание ввода-вывода (при наступлении события процессы переводятся из блокировки в состояние готовности)
 - Page fault (блок отсутствует в вирт. памяти -> ОС передает управление другому процессу, а старый переходит в блокировку до загрузки -> после загрузки переходит в состояние готовности)

- **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 выполнять 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.



Процесс начинает жить с системного вызова fork и в зависимости от наличия памяти попадает в состояние Ready в swap или память.

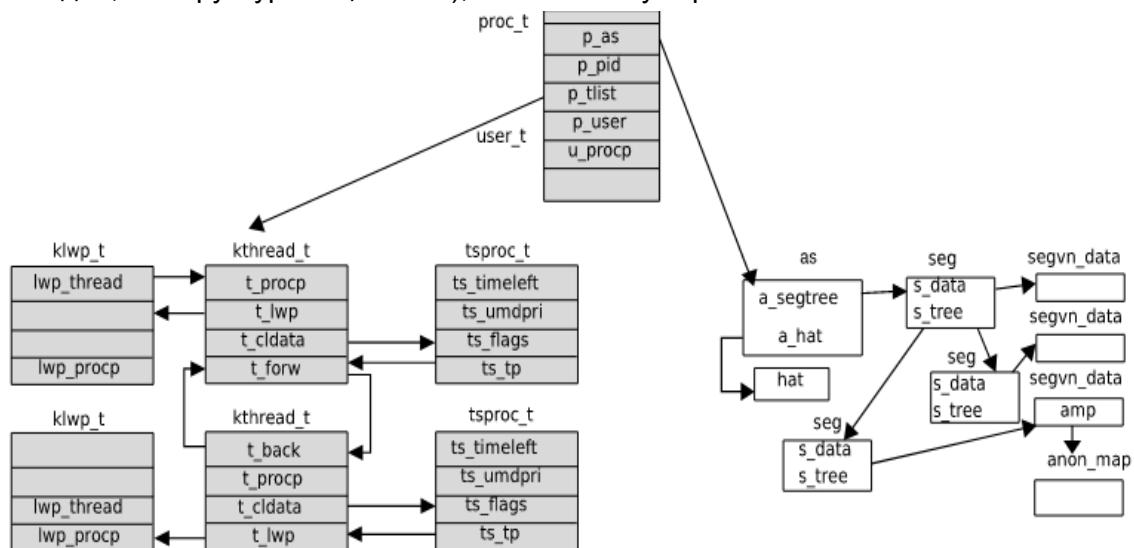
Два состояния исполнения: User Running и Kernel Running, связаны прерываниями (Syscall, Interrupt) и их возвратом (Return).

В ядре процессы тоже прерываются, при этом у них могут меняться контексты.

Во время диспетчеризации программу могут выкинуть в Runnable или вытеснить (Preempt), а на её место записать другой процесс.

Когда до процесса доходит очередь, он возвращается из состояния ожидания в рабочий режим.

При системном вызове Exit процесс попадает в состояние Zombie (когда он уже невалиден, но структуры еще живы), после чего “умирает”.

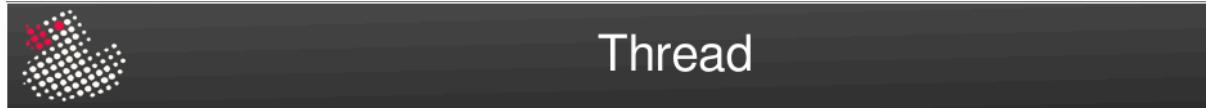


PCB в Solaris состоит из структур proc_t и user_t и занимает порядка 2 Кбайт памяти. p_as - указатель на адресацию памяти; содержит структуры, которые относятся к АП и его конкретным сегментам, а также определяют, именованное оно (сегмент данных в файле) или анонимное (куча, стек).

Также он содержит трэды на уровне ядра и пользователя.

29. Понятие потока выполнения, связь потока и процесса.

Преимущества потоков.



- Абстрактная модель процесса подразумевает:
 - Владение ресурсами (сегменты памяти, файлы, каналы ввода вывода, контекст безопасности, ...)
 - Планирование и диспетчеризацию (приоритеты, состояния) независимы
- Процесс — единица группировки общих ресурсов
- Thread (нить выполнения) — единица выполнения программного кода.
Содержит:
 - Состояние выполнения
 - Сохраненный контекст потока (регистры, ...)
 - Стек (ядра и пользователя)
 - Локальные переменные (thread_locals)
 - Доступ к памяти и другим ресурсам процесса-владельца

Свойства концепции процесса:

- 1) Владение ресурсами - образ процесса (программа, данные, стек, атрибуты PCB), основная память, IO-каналы и устройства и файлы).
- 2) Планирование/диспетчеризация - процессы при выполнении чередуются между собой, поэтому имеют состояния (след. слайд) и приоритет диспетчеризации.

Являются независимыми -> ОС может рассматривать их раздельно -> Процесс и поток

Процесс (задание) - единица группировки ресурсов.

Поток (легковесный процесс) - единица выполнения программного кода.

Различные ОС похожи друг на друга, но внутри программной модели могут достаточно сильно различаться.

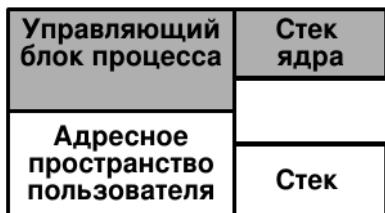
Характеристики потока:

- состояние выполнения (выполняется, готов, ...);
- сохраненный контекст (если поток не выполняется; регистры и т.п.);
- стек выполнения (ядра и юзера);
- локальные переменные;
- доступ к памяти и другим ресурсам процесса-владельца (разд. между всеми потоками).

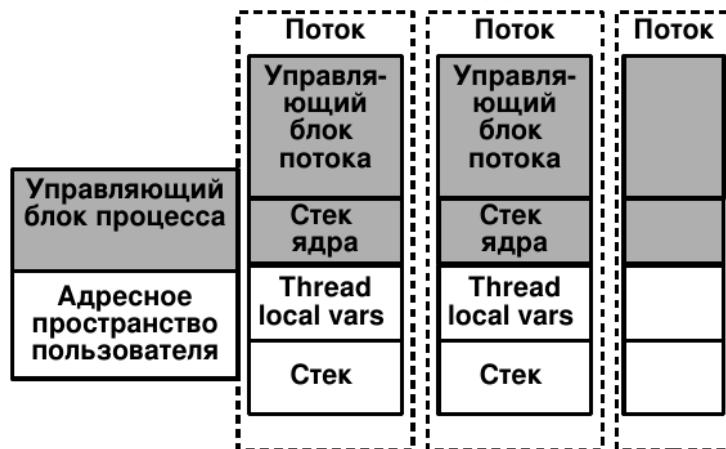


Связь структур ядра процесса и потока

Однопоточная модель



Многопоточная модель



Операционные системы. Часть 2. Процессы и потоки

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

Серый - ядро, белый - юзер.

Однопоточная модель - PCB (упр. блок процесса), АП юзера и стеки юзера и ядра (вызовы процедур и возвраты). При исполнении процесс контролирует регистры процессора, при остановке они сохраняются.

Многопоточная модель - PCB, АП юзера; для каждого потока - локальные переменные, стеки юзера и ядра, TCB (упр. блок потока) (значения регистров, приоритет и т.д.)

АП пользователя и сегменты кода и данных являются общими. Стеки связаны с контекстом исполнения и регистрами -> отдельные.

Структуры исполнения разные, ресурсы общие -> Конкуренция потоков за ресурсы



Преимущество потоков

- Общее преимущество 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

Преимущества по сравнению с процессами:

- 1) Для создания процесса нужно читать программы с диска и выделять память, а у потока программа уже загружена и АП юзера создано -> Потоки создаются на порядок быстрее процессов
- 2) Для переключения процесса нужно перезагружать много информации (например, АП), а для потока достаточно загрузить регистры -> Переключаются тоже быстрее
- 3) Для завершения процесса нужно закрыть связанные ресурсы (сегменты памяти, файлы), до считывания родителем кода возврата он пребывает в состоянии "зомби"; потоки же разделяют ресурсы -> Завершаются тоже быстрее
- 4) Обмен информацией между процессами сопровождается участием ядра и большими накладными расходами (смена контекста ядра, сиксколл, переключение процесса), а между потоками просто доступом к оперативной памяти без ядра -> Обмениваются информацией тоже быстрее

Преимущества:

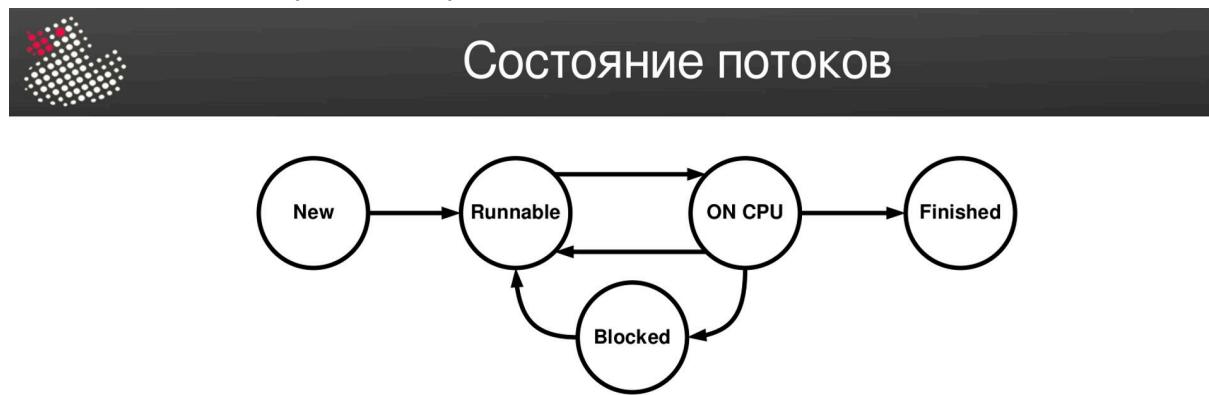
- 1) Скорость выполнения (вычисления над одной порцией данных параллельно с вводом других) (Только для многопроцессорных)
- 2) Работа в приоритетном и фоновом режимах (программа для электронных таблиц, один поток отображает меню и считывает ввод юзера, другой выполняет его команды и обновляет таблицу -> можно начинать ввод следующей команды до завершения предыдущей)
- 3) Асинхронная обработка частей программы (поток текстового редактора планируется непосредственно ОС и ежеминутно сбрасывает буфер ОЗУ на диск)
- 4) Модульная структура программы (разнообразные действия или множество ИО между разными источниками легче разрабатывать и реализовывать с помощью потоков)

Пример удачного использования потока - файловый сервер (поток под каждый запрос, на многопроцессорной машине может выполняться несколько потоков, совместное использование данных из файлов -> координация действий).

Приостановка процесса -> АП выгружается из памяти -> Приостановка потоков

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

Билет не расписан, пусть пока будет так.



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

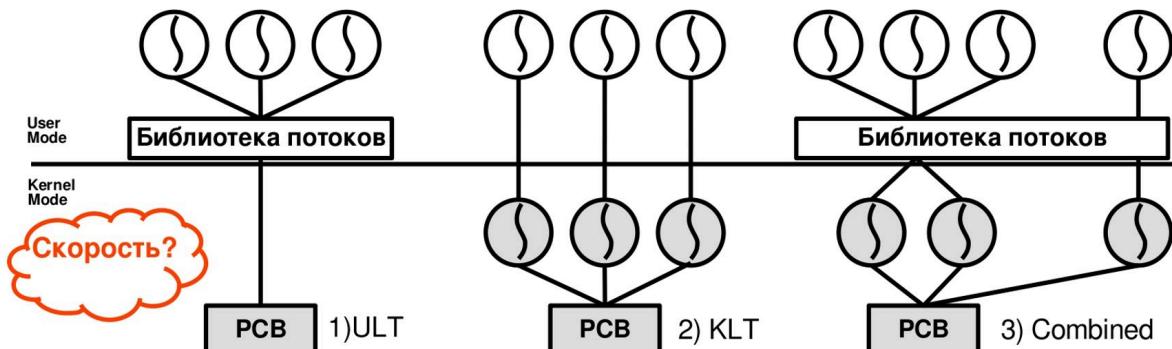
Похоже на состояния процессов без пейджинга и своппинга.

Сначала поток создаётся, потом попадает в очередь на исполнение, по приходу своей очереди - на CPU, откуда он может по прошествии кванта времени вернуться в очередь или оказаться заблокированным, например, вводом-выводом. После выполнения всех инструкций он может завершиться, а его структуры - пропасть. Т.к. потоки используют АП процесса, необходима синхронизация по общим данным. При этом блокирование потока не должно в общем случае блокировать процесс.



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



Операционные системы. Часть 2. Процессы и потоки

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

- 1) User Level Threads (Green Threads) - переключение между потоками не требует выхода в ядро. Операции реализуются спец. библиотеками (или приложениями) на стороне пользователя. Пример - Котлин и корутины.
- 2) Kernel Level Threads - легковесные процессы, трэды внутри ядра, которые сами по себе диспетчеризуются на процессоре. Обычно пользовательскому потоку соответствует LWP.
- 3) Т.к. необязательно иметь для каждого польз. трэда ядерные структуры, существует смешанный подход, при котором м.б. несколько диспетчеризуемых на процессоре потоков, т.е. треду на уровне ядра может соответствовать несколько пользовательских.

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.

Многопроцессорность - управление несколькими процессами в многопроцессорной системе.

Закон Амдала определяет, насколько можно ускорить выполнение в многопроцессорной системе с использованием потоков.



Многопроцессорность и многопоточность

- Насколько можно ускорить программу на N процессорах с использованием потоков?
- Закон Амдала:

$$\text{Ускорение} = \frac{\text{Время работы на одном процессоре}}{\text{Время выполнения на } N \text{ процессорах}} = \frac{T \times (1-f) + T \times f}{T \times (1-f) + \frac{T \times f}{N}} = \frac{1}{(1-f) + \frac{f}{N}}$$

где, Т - время работы; f — доля распараллеливания [0..1])

- Когда f мало, использование параллельного выполнения неэффективно
- Когда N → ∞, то ускорение ограничено 1/(1-f)

Операционные системы. Часть 2. Процессы и потоки

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Рассмотрим связь многопроцессорности и многопоточности.

В явном виде задача ставится следующим образом: на сколько можно ускорить программу, если ее распараллелить на несколько процессоров с использованием потоков?

Этой проблемой некоторое время назад задался Джим Амдал, который в 1967 году смог рассчитать это ускорение, предположив, что указанное ускорение можно определить, поделив время выполнения программы на одном процессоре на время выполнения программы на N процессорах.

В результате этих рассуждений родился так называемый закон Амдала, приведенный на слайде.

В предложенном законом формуле есть время работы программы T и есть параметр f, который говорит о том, как эффективно программа распараллелена (если f=0, то программа выполняется чисто последовательно, если f=1, то все части программы могут быть выполнены параллельно).

Обычно «в реальной жизни» f колеблется от 0.2 до 0.8.

f=0.9 может быть в серверах, обеспечивающих работу систем с большим количеством однотипных пользовательских запросов.

При помощи достаточно простых выводов, когда все время подразделяется на распараллеливаемое и не распараллеливаемое, и при учете, что распараллеливаемое время может выполняться на N процессорах, в результате получается формула закона Амдала.

Из закона Амдала видно, что если f - маленькое, то создавать потоки для параллельного выполнения оказывается неэффективно, потому что получаемое ускорение все равно мало.

Если увеличить количество процессоров до бесконечности, можно построить «максимально параллельную систему». Но как бы разработчики не «параллелили» программу, максимальное ускорение, которое можно получить будет 1/(1-f).

Если распараллеливание 0.99, то масштабирование имеет смысл.

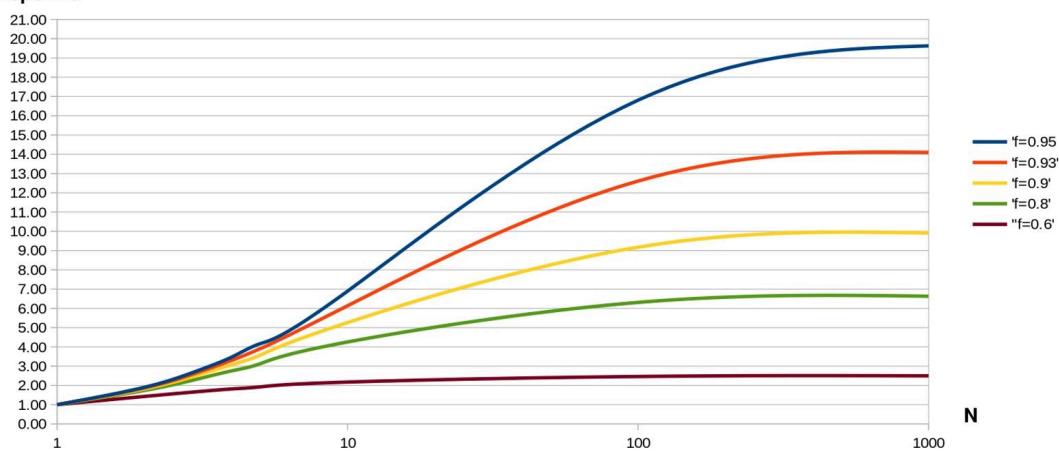
Если распараллеливание меньше, чем 0.99, то эффективность масштабирования остается «под вопросом», что очень хорошо видно на графике, который приведен на следующем слайде.

на синхронизации висеть можем



Закон Амдала

Ускорение



Операционные системы. Часть 2. Процессы и потоки

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

На графике представлена зависимость ускорения «в разах» от количества процессоров, рассчитанная согласно закону Амдала.

Из приведенных зависимостей, например, следует, что при $f=0.95$ (когда практически вся наша программа является «параллельной») при количестве процессоров в 1000 можно получить ускорение всего в 20 раз.

Таким образом стоит рассмотреть вопрос: а стоит ли игра свеч?

Из графиков видно, что на начальных участках ускорение достаточно быстро растет. И на 10 процессорах ускорение будет в 7 раз, а при 100 процессорах ускорение будет всего в 17 раз.

Поэтому даже серьезно увеличивая f и приближая его к единице, никогда не удастся получить во столько раз увеличения производительности, во сколько раз было увеличено количество процессоров (в связи с тем, что у любых программ есть «общее время», необходимое в том числе и на синхронизацию, которая не поддается распараллеливанию).

Рассмотрим другие «вырожденные» ситуации, когда у нас, например, $f=0.6$. И получается, что при 1000 процессоров ускорение происходит всего в 2.5 раза.

Это надо иметь в виду, когда разрабатываются «параллельные» программы.

В случае, если рассматриваемый сервер уже запущился, то начальную часть его работы можно «отбросить», и она не будет существенным образом влиять на «поведение» закономерностей. И тогда на эффективность распараллеливания начинают сильно влиять те, части потоков, которые необходимо синхронизировать: на сколько «крупные» куски потоков синхронизируются, как много времени уходит на их синхронизацию. Именно это и будет влиять на коэффициент f .

Следует отметить, что именно из этих рассуждений родились разработки «неблокирующих алгоритмов», позволяющих добиваться значительного снижения времени на синхронизацию. Однако, стремление повышать производительность путем повышения степени параллельности, описываемой коэффициентом f , не всегда удается реализовать в полном объеме.

32. Механизм параллельных вычислений, функции ОС.

- Однопроцессорные системы — процессы чередуются



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



Диспетчер - выбирает процессы, которые пойдут на исполнение.

При наличии в системе только одного процессора мы можем реализовать псевдопараллелизм, посылая на процессор разные потоки чередуя их друг с другом. Если мы имеем несколько процессоров, то мы можем добиться истинного параллелизма на уровне задач для 1 процесса.

На картинке ниже процесс 3 занимает так много времени из-за приоритетов или локальности (чтобы кеши туда сюда не гонять). Это и есть одна из причин почему привязка процесса к процессору может повысить производительность.

Один из возможных ситуаций, когда может наступить голодание:

- низкоприоритетный процесс захватил блокировку и не дает выполниться высокоприоритетному

Основные механизмы

- Квант времени на процессоре
- Приоритет задач
- Механизм очереди потоков на процессор

Требуемые функции ОС:

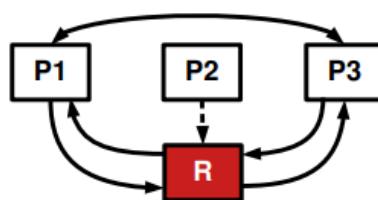
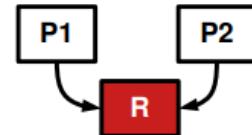
- Отслеживание ресурсов процесса/потока
- Распределение и освобождение ресурсов для каждого активного процесса/потока (CPU, файлы, память, ...)
- Защита ресурсов процесса/потока от непреднамеренного воздействия других процессов/потоков
- Независимость результата процесса/потока от скорости его выполнения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

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



Проблемы параллельного выполнения

- Взаимоисключений (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



Операционные системы. Часть 2. Процессы и потоки
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Требования к взаимным исключениям

- Взаимоисключения осуществляются в принудительном порядке.
 - В критическом участке кода должен находиться 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

Проще будет запомнить в таком виде: заходит мистер поток в бар (а может и забегает, **скорость не важна(5)**) и говорит бармену: “налей ка мне критического ресурса”. Бармен ему отвечает: “извини, **без блокировки не обслуживаем(2)**..... пить его может **только один поток(1)**, дождись его, но будь уверен, **бесконечно ждать не будешь(3)**, я тебя **сразу позову(4)**, а пока пойду напомню ему что **его время кончается(6)**”.

- **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 требовать доступ к одному и тому же диску или к принтеру. Операционная система должна регулировать такие обращения.

Взаимоисключения, взаимоблокировки, голодание

- **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.

Взаимоисключения, взаимоблокировки, голодание, связь данных (когда один поток испортит данные общего ресурса)

- **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.

Взаимоблокировки, голодание (взаимоисключений нет, так как потоки всё таки что то знают друг о друге и среди работы). Пример - СУБД Oracle, где много потоков работают в среде с разделенной памятью, и постоянно и согласованно меняют состояние памяти.

34. Примитивы синхронизации ОС. Предназначение примитивов синхронизации



Основные примитивы

- Семафоры (Semaphore) — захват и освобождение множественного ресурса
 - или одного (бинарные семафоры)
- Мьютексы (Mutex) — блокировка и освобождение ресурса единственным процессом/потоком
- Условные переменные (Conditional Variable) — Блокировка до выполнения какого-либо условия
- Блокировки чтения/записи (rw-lock) — отдельные блокировки на чтение и запись
- 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

Столлингс гл. 5.1 — Эволюция подхода к блокировке — самостоятельно прочитать!

+ СПИНКЛОКИ

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)

290 ГЛАВА 5. ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ: ВЗАЙМОИСКЛЮЧЕНИЯ И МНОГОЗАДАЧНОСТЬ

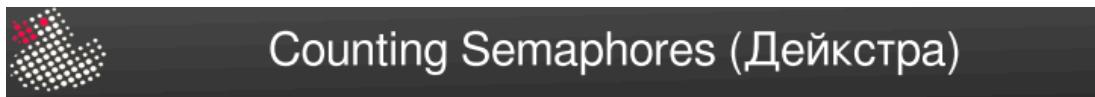
Окончание табл. 5.3

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 <i>критические участки</i> . 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

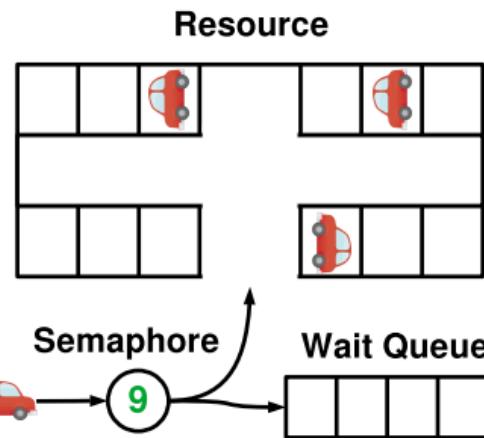
35. Примитивы синхронизации ОС. Семафоры и мьютексы.

Бинарный семафор

1. **Семафоры** (Semaphore) - захват и освобождение множественного ресурса (или одного - бинарные семафоры). Блокирующий примитив



- sema_p() «proberen», semWait
 - count--
 - if count<0
 thread blocks in WQ
- sema_v() «verhogen», semSignal
 - count++
 - if count<=0
 notify thread
- Политика выборки разная
 - строгая, слабая, приоритет



“proberen” (проверка) - уменьшаем счётчик ресурса и проверяем: если count<0 - блокируем поток в очереди, иначе - допускаем к ресурсу.

“verhogen” - когда ресурс освобождается потоком, проверяем, если ли в очереди потоки, если есть - запускаем поток из очереди

Потоки, которые находятся в очереди - именно блокируются (состояние ожидания), поэтому очень дорогой и медленный способ (сначала переключение контекста при блокировании, затем обратно возвращаем контекст при извлечении из очереди - достаточно ресурсозатратно).

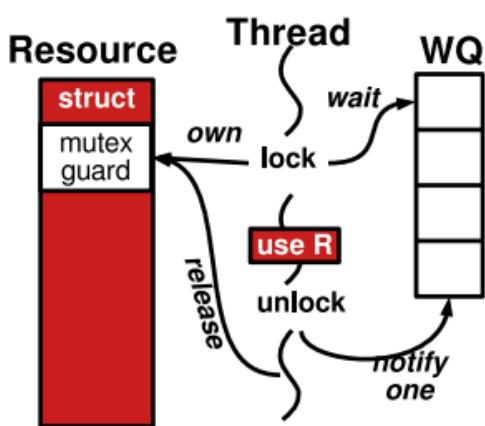
Используются для счётных ресурсов (например, параллельное чтение, пул работников (workers))

В Linux up(), down().

2. **Мьютексы** (Mutex) - блокировка и освобождение ресурса единственным потоком/процессом. Т.е. какой поток захватил ресурс, этот же поток и должен его освободить (другим потокам доступ закрыт, они не могут войти в ресурс / снять блокировку). Обеспечивает взаимное исключение исполнения критических участков кода.
lock(), unlock()



Mutexes



- Множество различных реализаций:
 - Блокирующие, спин, адаптивные, фьютексы, ...
- Не- бинарный семафор!
 - Захвативший блокировку должен ее освободить
- Захват должен быть коротким
- Priority inversion

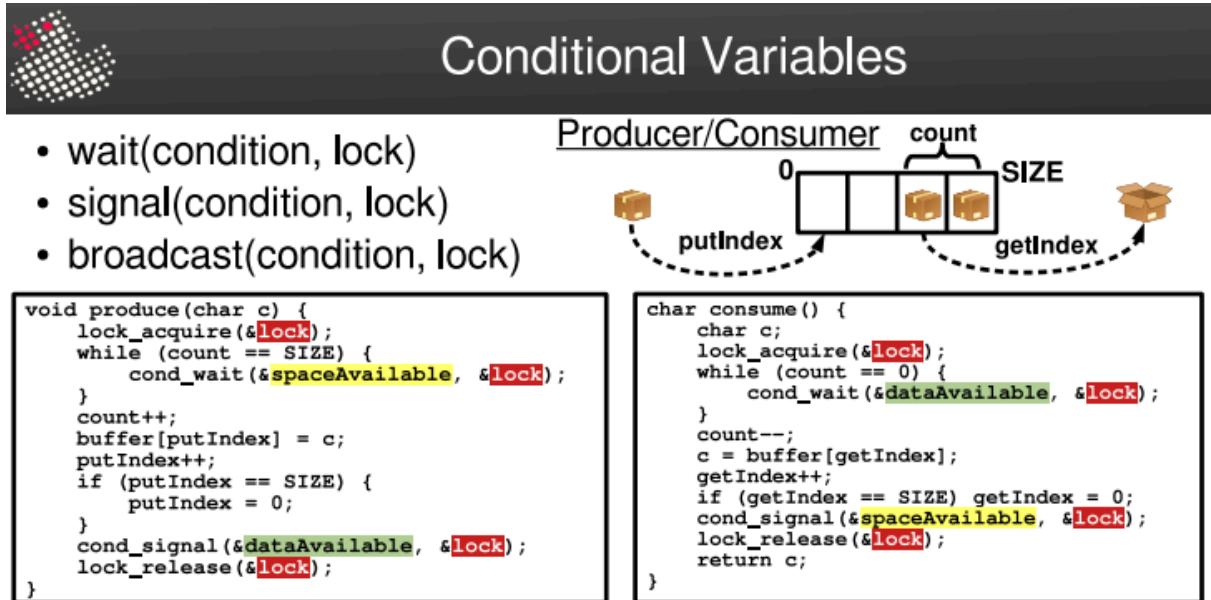
Могут быть реализованы по-разному: фьютексы - UserSpace мьютексы, которые стараются как можно меньше блокировать, потому что блокировка вызывает смену контекста ядра; блокирующие, адаптивные, напр если тот кого он ожидает заблокирован, то переходим в wait, иначе спинимся и т.д.

Проблема(priority inversion): если поток с низким приоритетом захватил ресурс, пришедший поток с высоким приоритетом (realTime) не сможет вытеснить низкий поток и занять ресурс, он также отправится в очередь (а это не очень хорошо).

3. **Спин-локи (Spinlock)** - циклическая блокировка. На уровне ядра они эквивалентны одной из разновидностей мьютекса. Применяются когда ожидание захвата блокировки предполагается недолгим, либо если контекст выполнения не позволяет переходить в заблокированное состояние.

36. Примитивы синхронизации ОС. Условные переменные, rwlocks.

1. **Условные переменные** (Conditional Variable) - блокировка до выполнения какого-либо условия.



Блокирование до момента поступления сигнала от другого потока о выполнении (wait, notify, notifyAll).

signal - пробуждает один (какой-то), broadcast - пробуждает всех кто ждет. wait - ждать наступления условия.

2. **Блокировки чтения/записи** (rw-lock) - отдельные блокировки на чтение и запись. Основная задача: сделать так, чтобы много потоков могли одновременно читать, и только один писать. То есть когда ресурс не заблокирован, у него может быть произвольное количество читателей. Но когда кому-то захочется что-то записать, он посылает сигнал на запись, и ждет, когда все читатели уйдут с ресурса. Как только это наступает, на запись допускается один писатель и по завершению отсылает всем сигнал, что ресурс снова свободен на чтение.



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)



Проблема “громящего стада” - когда ресурс освобождается (дается разрешение на запись) все потоки хотят максимально быстро захватить этот ресурс, кто первый успел - тот молодец (погоня за ресурсом).

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

1. **Мониторы** (Monitor) - конструкции ЯП, которые скрывают низкоуровневые примитивы синхронизации. Например, в Java есть ключевое слово *synchronized* - некое подобие реализации монитора



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

Плюсы: это просто (для программиста)

Минусы: программист может не понимать всю суть работы этого метода и пихать мониторы везде, где захочется (переизбыток будет тормозить систему).
Эффективнее использовать более низкоуровневые примитивы.

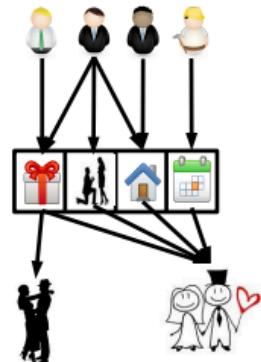
2. **Флаги событий** (Event Flags) - связывание условий продолжения выполнения с одним или несколькими флагами (битами блокирующей переменной). Является

редким примитивом блокировки, используется в старых системах, также есть в Python. Суть: кто-то ставит флаги (о вводе/выводе/готовности каких-то частей), другой поток может прочитать значение этих флагов и продолжить выполнение, если установлены определенные флаги.



Event Flags

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



VMS - операционная система.

3. Почтовые ящики (Message Passing) - примитив передачи сообщений.

Высокоуровневый примитив, достаточно часто используется в очередях.



Message passing

- Две операции — send(получатель, сообщение) и receive(отправитель, сообщение)
- Необходим метод адресации получателя и отправителя
 - Прямая адресация (direct): явная и постфактум
 - Косвенная адресация (indirect) — номер «почтового ящика» - 1:1, 1:N, M:1, M:N
- Раздельная синхронизация посылки и получения
 - Блокирующая, неблокирующая, гарантия доставки
- Формат сообщения
 - Фиксированная, переменная длина, файл, ...
- Выборка из очереди — Приоритет, FIFO, ...
 - ожидаем сообщения либо от определенного отправителя, или узнаем уже после того, как это сообщение придет

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

Блокировка вызывает переключение контекста - долго (дорого)!

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 простоявая. По максимуму — возьмут невыполненную работу на себя.

Типы неблокирующих алгоритмов:

1. **Wait-free** (N-steps) (Без ожиданий)

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

Метод свободен от ожидания (**wait-free**), если **каждый** вызов этого метода завершается за **конечное** число шагов, вне зависимости от поведения других потоков.

Свобода от ожидания гарантирует прогресс **каждому** вызову метода.

2. **Lock-free** (come N-steps, other retry) (Без блокировок)

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.

Не строгое, зато интуитивно понятное определение:

Метод свободен от блокировок (**lock-free**), если хотя бы один из вызовов метода продвигается вперед, т.е. имеет место **глобальный прогресс**, вне зависимости от поведения других потоков.

3. **Obstruction-free** (Без препятствий)

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 простоявать.

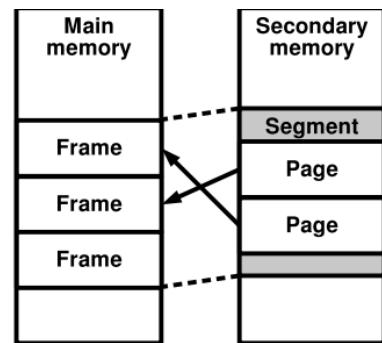
Гарантия, интересная только теоретикам:

Назовем метод **obstruction-free**, если он завершается за конечное число шагов в ситуации, когда конкурирующие вызовы не исполняются.

Неблокирующие структуры: атомарные операции и примитивы синхронизации (спин-лок блокировка), различные реализации списков, деревьев, очередей...
<https://habr.com/ru/post/219201/>

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

- Основная память (main memory)
 - Область, где процессор может исполнять программы
- Вторичная память (secondary memory)
 - Область, где программы и данные могут храниться, в том числе и во время выполнения
- Кадр (frame) — область фиксированного размера основной памяти
- Страница (page) — область фиксированного размера во вторичной памяти, которая может быть скопирована в кадр
- Сегмент (segment) — область переменного размера в основной или вторичной памяти, может быть разделена на страницы



Требования (Столлингс, глава 7.1, написано замечательно):

- **Переместимость**
Неизвестно, где именно будет размещена программа. Неизвестно, куда она будет перемещена при свопинге. Организация памяти ОС должна позволять программе функционировать вне зависимости от перемещений
как я понимаю, сейчас это везде достигается виртуализацией. программа оперирует виртуальными адресами, которые ось может мапить в любые физические
- **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. Для этого необходимо проверять обращения к памяти в рантайме. Более того, это должно делаться аппаратно
- **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.

Система управления памятью должна, таким образом, обеспечивать управляемый доступ к разделяемым областям памяти, при этом никоим образом не ослабляя защиту памяти.

- **Логическая организация**

Управление основной и вторичной памятью

Поддержка модульной организации программ: нам не важно единое линейное адресное пространство для всей программы, но важно для каждого модуля

- **Физическая организация**

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, но обеспечивает долговременное хранение.

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.

- a. Основной памяти может быть недостаточно для программы и ее данных.

В этом случае программист вынужден прибегнуть к практике, известной как структуры с перекрытием - оверлэй (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 оверлееев разработка таких программ приводит к дополнительным затратам времени программиста.

- b. В многозадачной среде программист при разработке программы не знает, какой объем памяти будет доступен программе и где эта память будет располагаться.

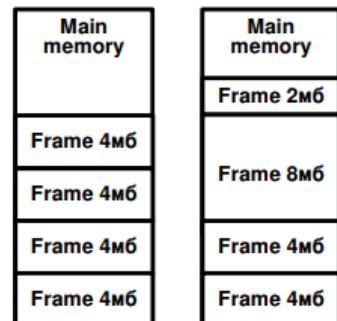
Таким образом, очевидно, что задача перемещения информации между двумя уровнями памяти должна возлагаться на ОС

40. Фиксированное и динамическое размещение программ в памяти.

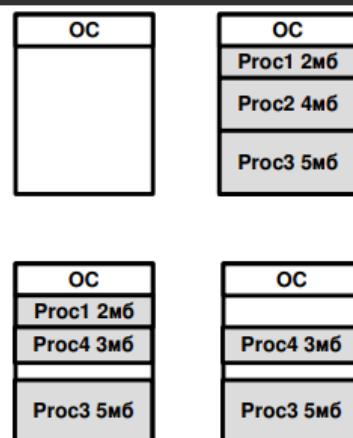


Fixed partitioning

- У программ разная длина, их сложно упаковывать в память
- А что если оперировать кадрами заданного размера?
- Минусы
 - Память расходуется неэффективно для небольших процессов
 - Ограничение на количество одновременных процессов
 - Если процесс не помещается в раздел, он должен использовать перекрытия (overlays), самостоятельно загружая и выгружая свои части

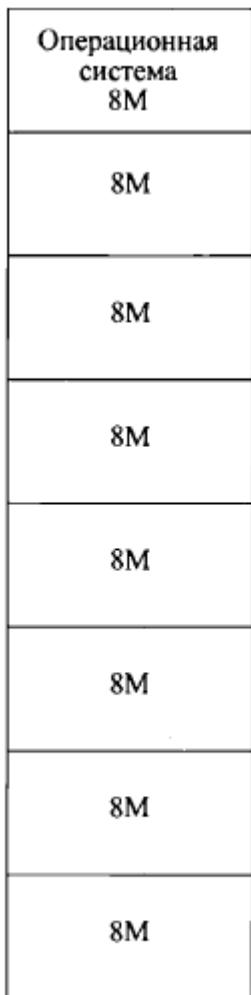


Dynamic partitioning

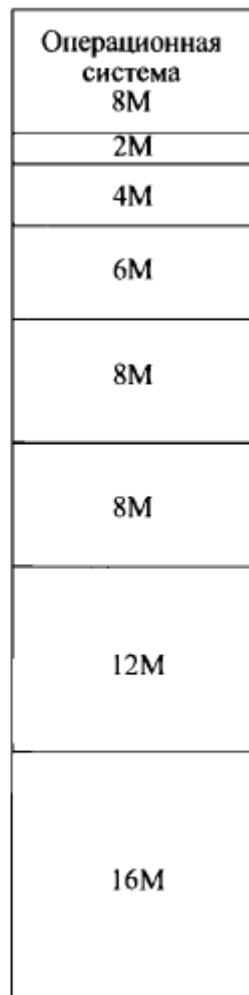


- Блоки процессов разного размера, ОС встраивает их в свободные места
- Минусы
 - Фрагментация памяти
 - Необходимость упаковки памяти
 - Начинает за здравие, заканчивает ну вы понимаете...
 - Сложные и медленные алгоритмы размещения в памяти
- Подходит для загрузки драйверов
 - Поэтому и нужна была перезагрузка при добавлении драйверов
- Разумный компромисс - «Buddy System» - см. Столлингса.

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 страницчная 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 разделен 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
Страницчная организация виртуальной памяти	Все, как при простой страницочной 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



а) Разделы одинакового размера



б) Разделы разных размеров

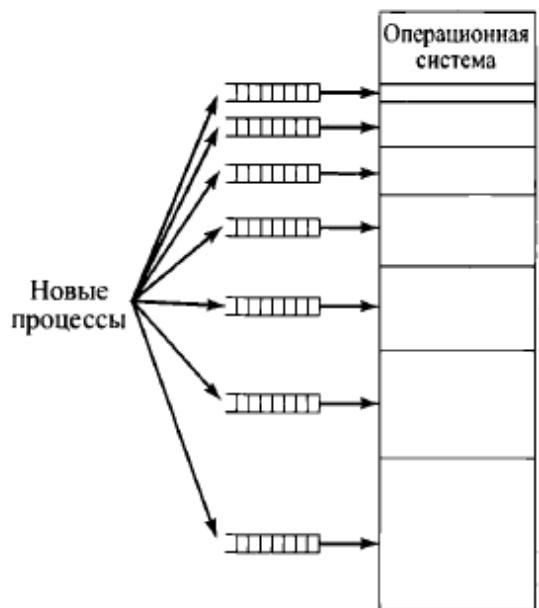
Алгоритм размещения

В том случае, когда разделы имеют одинаковый размер, размещение процессов в памяти представляет собой тривиальную задачу. Не имеет значения, в каком из свободных разделов будет размещен процесс. Если все разделы заняты процессами, которые не готовы к немедленной работе, любой из них может быть выгружен для освобождения памяти для нового процесса. Принятие решения о том, какой именно процесс следует выгрузить, — задача планировщика (об этом мы поговорим в части IV, “Планирование”).

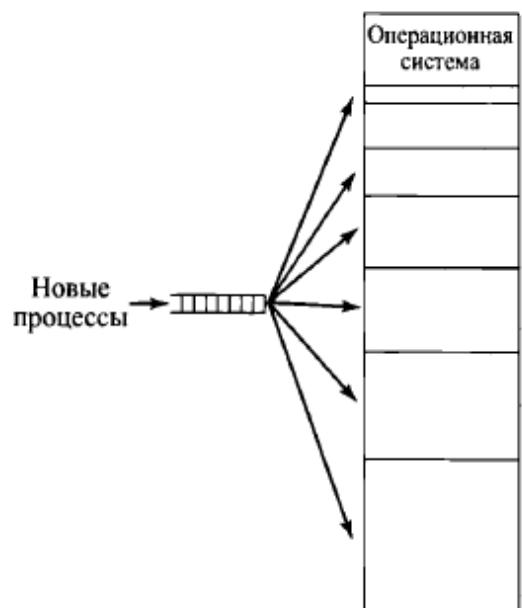
Когда разделы имеют разные размеры, есть два возможных подхода к назначению процессов разделам памяти. Простейший путь состоит в том, чтобы каждый процесс размещался в наименьшем разделе, способном полностью вместить данный процесс.¹ В таком случае для каждого раздела требуется очередь планировщика, в которой хранятся выгруженные из памяти процессы, предназначенные для данного раздела памяти (рис. 7.3, а). Преимущество такого подхода заключается в том, что процессы могут быть распределены между разделами памяти так, чтобы минимизировать внутреннюю фрагментацию.

Хотя этот метод представляется оптимальным с точки зрения отдельного раздела, он не оптimalен с точки зрения системы в целом. Представим, что в системе, изображенной на рис. 7.2, б, в некоторый момент времени нет ни одного процесса размером от 12 до 16 Мбайт. В результате раздел размером 16 Мбайт будет пустовать, в то время как он мог бы с успехом использоваться меньшими процессами. Таким образом, более предпочтительным подходом является использование одной очереди для всех процессов (см. рис. 7.3, б). В момент, когда требуется загрузить процесс в основную память, для этого выбирается наименьший доступный раздел, способный вместить данный процесс. Если все разделы заняты, следует принять решение об освобождении одного из них. Понятно, следует отдать предпочтение процессу, занимающему наименьший раздел, способный вместить загружаемый процесс. Можно учесть и другие факторы, такие как приоритет процесса или его состояние (заблокирован он или активен).

Использование разделов разного размера по сравнению с использованием разделов одинакового размера придает дополнительную гибкость данному методу. Кроме того, схемы с фиксированными разделами относительно просты, предъявляют минимальные требования к операционной системе; накладные расходы работы процессора невелики.



а) Отдельная очередь процессов
для каждого раздела

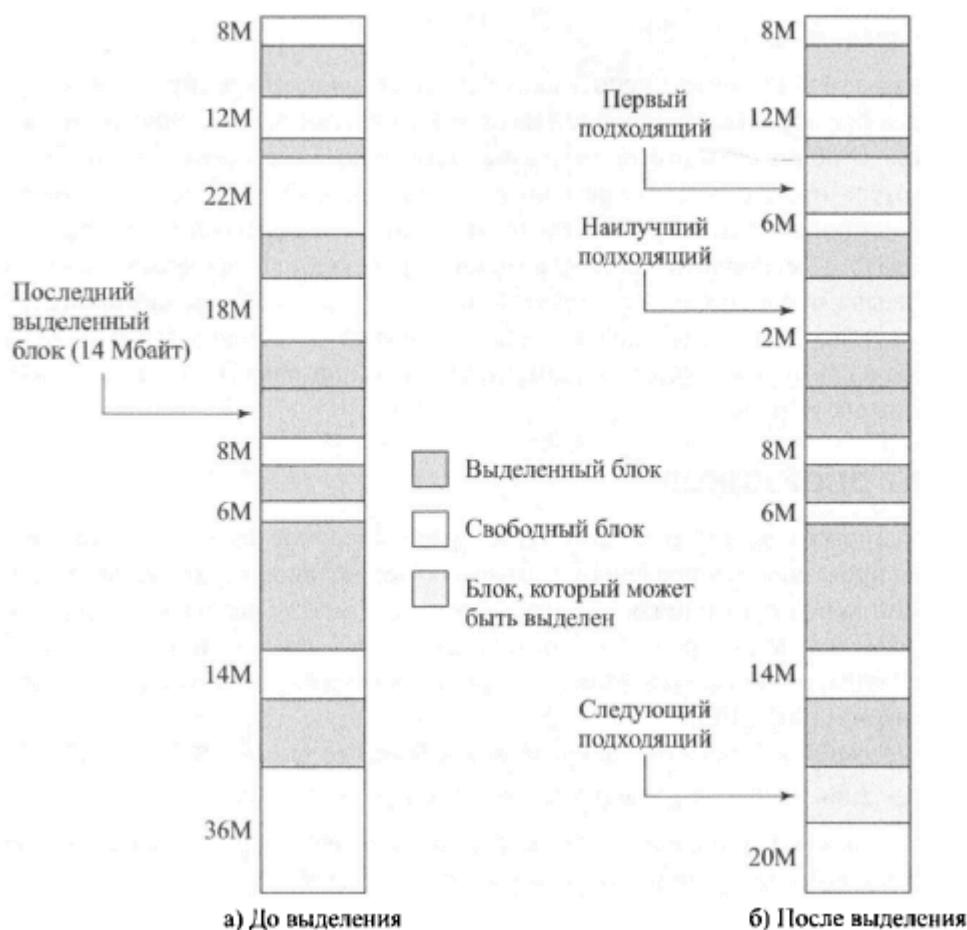


б) Общая очередь для всех
процессов

Поскольку уплотнение памяти вызывает дополнительные расходы времени процессора, разработчик операционной системы должен принять разумное решение о том, каким образом размещать процессы в памяти (образно говоря, каким образом затыкать дыры). Когда наступает момент загрузки процесса в основную память и имеется несколько блоков свободной памяти достаточного размера, операционная система должна принять решение о том, какой именно свободный блок использовать.²

Можно рассматривать три основных алгоритма — наилучший подходящий, первый подходящий, следующий подходящий. Все они, само собой разумеется, ограничены выбором среди свободных блоков размера, достаточно большого для размещения процесса. Метод **наилучшего подходящего** выбирает блок, размер которого наиболее близок к требуемому; метод **первого подходящего** проверяет все свободные блоки с начала памяти и выбирает первый достаточный по размеру для размещения процесса. Метод **следующего подходящего** работает так же, как и метод первого подходящего, однако начинает проверку с того места, где был выделен блок в последний раз (по достижении конца памяти он продолжает работу с ее начала).

7.2. РАСПРЕДЕЛЕНИЕ ПАМЯТИ 411



Какой из этих методов окажется наилучшим, будет зависеть от точной последовательности загрузки и выгрузки процессов и их размеров. Однако можно говорить о некоторых обобщенных выводах (см. [19], [28], [228]). Обычно алгоритм первого подходящего не только проще, но и быстрее и дает лучшие результаты. Алгоритм следующего подходящего, как правило, дает немного худшие результаты. Это связано с тем, что алгоритм следующего подходящего проявляет склонность к более частому выделению памяти из свободных блоков в конце памяти. В результате самые большие блоки свободной памяти (которые обычно располагаются в конце памяти) быстро разбиваются на меньшие фрагменты и, следовательно, при использовании метода следующего подходящего уплотнение должно выполняться чаще. С другой стороны, алгоритм первого подходящего обычно засоряет начало памяти небольшими свободными блоками, что приводит к увеличению времени поиска подходящего блока в последующем. Метод наилучшего подходящего, вопреки своему названию, оказывается, как правило, наихудшим. Так как он ищет блоки, наиболее близкие по размеру к требуемому, он оставляет после себя множество очень маленьких блоков. В результате, хотя при каждом выделении впустую тратится наименьшее возможное количество памяти, основная память очень быстро засоряется множеством мелких блоков, неспособных удовлетворить ни один запрос (так что при этом алгоритме уплотнение памяти должно выполняться значительно чаще).

Доп по вопросу - система двойников стр 412 столингс

41. Модели аппаратного перемещения программ.

(АХТУНГ!) 8086 в Столлингсе отсутствует

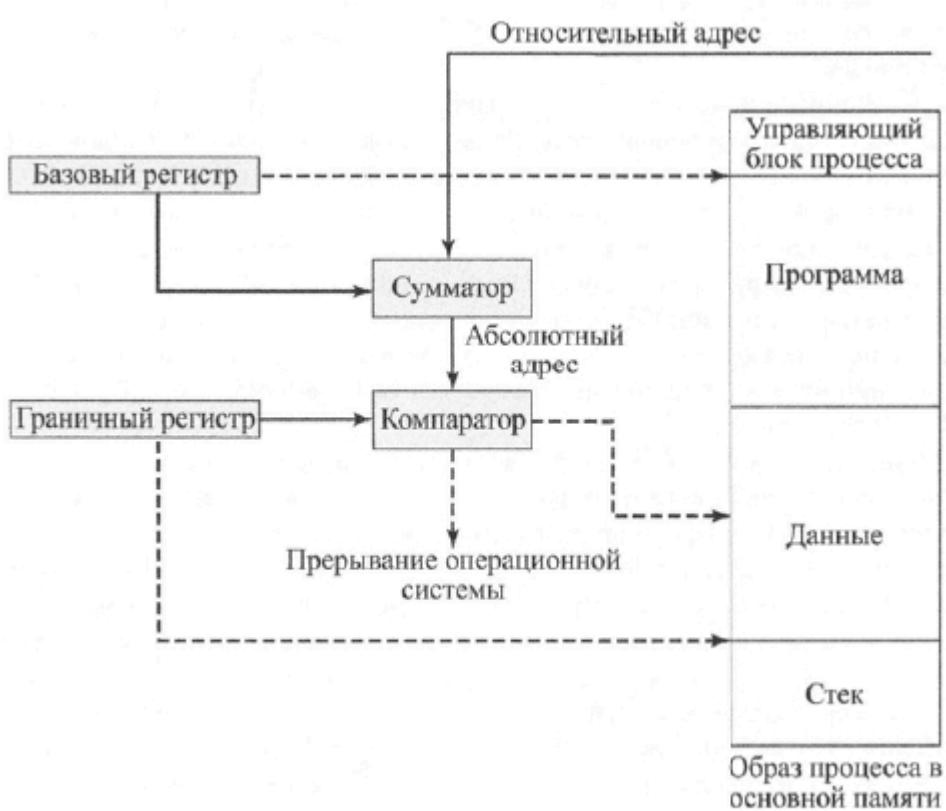


Рис. 7.8. Аппаратная поддержка перемещения

Перемещать программы средствами ОС или копирования блоков памяти очень долго. Необходимо помочь ОС работать с памятью средствами аппаратуры, включив туда некоторые доп. элементы.

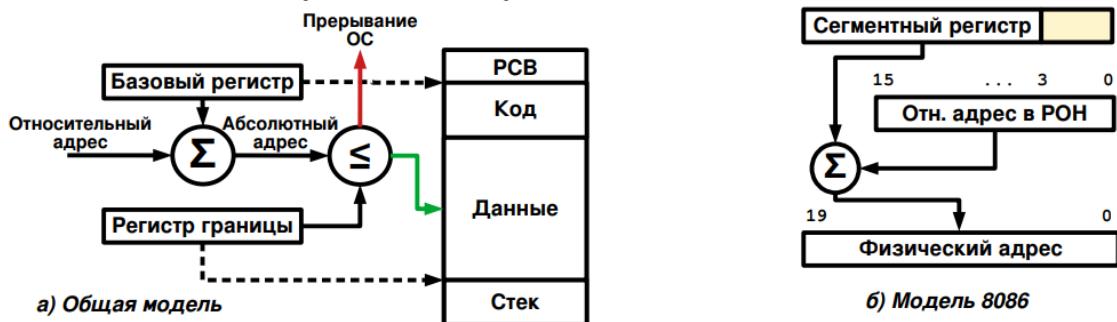
Классический случай приведен на левой части слайда. Он включает аппаратную поддержку для каждой программы двух регистров:

- базовый регистр - указывает на начало программы,
- регистр границы - конец программы (иногда вместо регистра границы используется длина сегмента).

В пользовательской программе относительный адрес закодирован в виде смещений. Он складывается с базовым регистром, в результате получается абсолютный адрес (зелёная стрелка).

Регистр границы обычно используется для проверки выхода абсолютного адреса за границы пользовательской программы (пунктир у границы стека). Выше базового регистра обратиться нельзя, ниже регистра границы тоже. При этом данные, необходимые для работы ОС, тоже находятся вне области пользовательской программы, например, PCB. При выходе за границы вызывается прерывание ОС.

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



8086: 16-разр. процессор \rightarrow адресует только $2^{16} = 64$ Кбайт; новый способ - $2^{20} = 1$ Мбайт

4 сегментных регистра: DS - Data Segment, CS - Code, SS - Stack, ES - Extra.

Сегмент имеет размер до 64 Кбайт и всегда начинается с адреса, кратного $2^4=16$, т.е. на границе 16-байтового блока памяти (параграфа). Поэтому сегментные регистры (16 бит) хранят нач. адрес сегмента без 4 младших битов. В РОН (регистр общего назначения) (тоже 16 бит) хран. относительный адрес (смещение отн. сегмента). Сегментный регистр сдвигается на 4 разряда и складывается с относительным адресом \rightarrow 20-разрядный физический адрес. При этом теоретически можно получить один ФА разными способами.

42. Простой страничный поход и простая сегментная организация.

Билет не расписан, пусть пока будет так.



- Давайте разделим память на кадры одинакового (небольшого) размера
 - Пусть страница занимает целиком кадр
 - Для удобства размер кратен степени 2
- Уменьшится внутренняя фрагментация
 - Пустое место будет в последнем блоке каждого процесса
- Внешняя фрагментация исчезнет совсем
- Необходимы таблицы страниц
 - ОС должна знать, где ее герои

Таблицы страниц	
Proc A	0
1	1
2	2
Proc B	3
6	4
7	5
8	6
Proc C	7
3	8
4	9
5	10
9	11
10	12
	...

Подход Simple paging заключается в предложении разделить память на кадры одинакового и достаточно небольшого размера. При этом страница будет занимать целиком кадр. Для удобства размер кадра должен быть кратен двум. А чтобы запустить процесс необходимо взять некоторое количество кадров.

Рассмотрим следующий пример. Пусть ПроцессA занимает 2 мегабайта - два кадра по 1Мбайту. (В том случае, если он будет иметь размер 1,5Мбайта, то он также займет два кадра основной памяти вычислительной системы).

При осуществлении Simple paging кроме всего прочего приходится поддерживать таблицу страниц. В этой таблице страниц находятся номера тех кадров, которые на данный момент заняты тем или иным процессом. Причем такие таблицы должны существовать для каждого процесса.

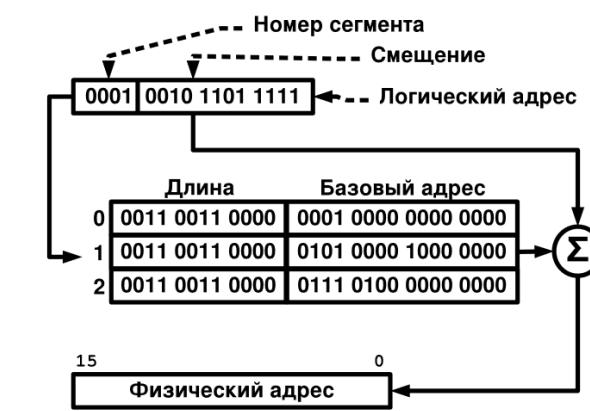
На приведенном рисунке показана ситуация того, что в какой-то момент между процессами A и B в память был загружен процесс C, часть которого загрузилась и «после процесса B».

Отсюда видно основное преимущество метода: операционная система, когда у нее память разрезана на странички малого размера, может эффективно размещать в памяти много процессов. При этом внешняя фрагментация в этом случае будет отсутствовать в принципе. А внутренняя фрагментация (если будет выбран достаточно маленький размер кадра) будет несущественна.

При реализации Simple paging необходимо, чтобы операционная система при каждом запуске нового процесса создавала для него собственную таблицу загрузки и соответствующим образом поддерживала таблицы загрузки всех страниц исполняемых процессов, при необходимости меняя их для того, чтобы поддерживать информацию о процессах в текущем состоянии.

Отличие этого метода Simple paging от Fixed Partitioning в том, что Fixed Partitioning предполагает неразрывность размещения текстов программ.

Simple Segmentation



- Таблица сегментов. Программист должен устанавливать:
 - Длину сегмента
 - Базовый адрес
- Внутренняя сегментация отсутствует
- Внешняя сегментация снижается

Еще одним методом работы с памятью является простая сегментация - Simple Segmentation.

«Простая сегментация» (Simple Segmentation) реализована с использованием таблицы сегментов. Причем эта таблица может находиться в регистрах или в оперативной памяти (чаще всего – в оперативной памяти).

В этой таблице описаны длина сегмента и некоторый базовый адрес. И когда приходит логический адрес программы, который состоит из смещения и номера сегмента, тогда операционная система (или соответствующая аппаратура), используя номер сегмента как индекс в этой таблице, берет из таблицы необходимый базовый адрес. После этого к нему прибавляет смещение и проверяет, чтобы получившийся адрес не выходил за размер сегмента. Так определяется физический адрес.

В адресном пространстве компьютера программы располагаются таким образом, что каждое расположение создает адрес в таблице. При этом получается, что внутренняя сегментация отсутствует, потому что сегмент занимает ровно столько, сколько нужно нашей программе. А внешняя сегментация серьезно снижается, и при этом нам не нужно упаковывать процессы.

Основное преимущество «простой сегментации» в том, что при ее использовании исчезает необходимость перемещать процессы. Достаточно лишь правильно указать их характеристики в таблице сегментов. После этого, процесс будет занимать столько места в памяти, сколько ему отвели.

43. Виртуальная память основные определения и принципы организации аппаратуры и управляемых 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

Resident set (резидентная часть) — часть процесса, находящаяся в основной памяти

Real memory — часть памяти, где процесс непосредственно выполняется

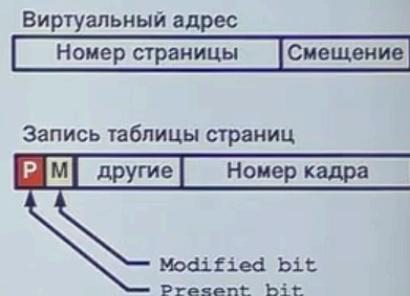
Virtual memory — часть памяти, которую процесс может занять

Организация виртуальной памяти – совокупность аппаратных и программных средств

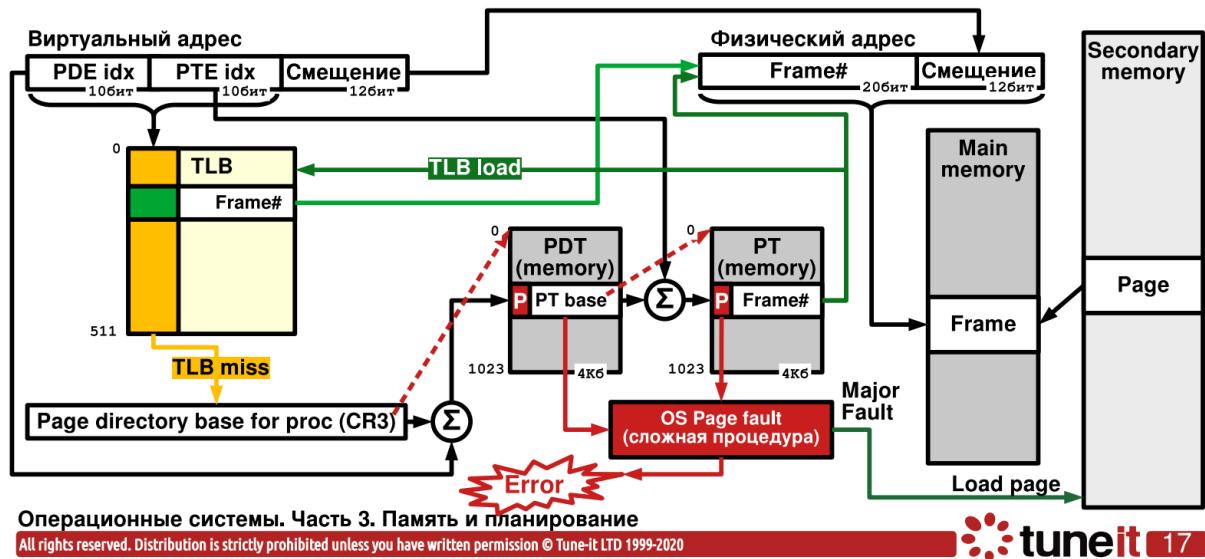
Схемы реализации:

- 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 находиться во вторичной памяти и менять физические адреса в основной

- Разные структуры организации
 - Одноуровневая
 - Двух- и многоуровневая
 - Инвертированная
- Таблицы страниц хранятся в основной памяти
 - Для обращения к памяти нужно сделать несколько служебных обращений к памяти
 - Используется TLB для ускорения



44. Виртуальный страничный обмен. Двухуровневая организация MMU и TLB 80386.



На этом фото:

- Виртуальный адрес (т.к. 2x уровневая память, то 2 индекса)
 - PDE idx (page directory entry) указывает на смещение в таблице PDT (данная таблица разбита на сегменты, сегмент выбирается с помощью CR3)
 - PTE idx (page table entry) указывает на смещение в таблице страниц
 - Смещение - смещение от начала страницы к нужному адресу
- Физический адрес: адрес страницы + смещение
- PDT - основная таблица размещения процессов
- PT - таблица страниц, выбирается номер страницы
- CR3 - регистр, который указывает на начало PDT (уникален для каждого процесса)
- TLB - хранит последние запросы страниц

Алгоритм поиска страницы:

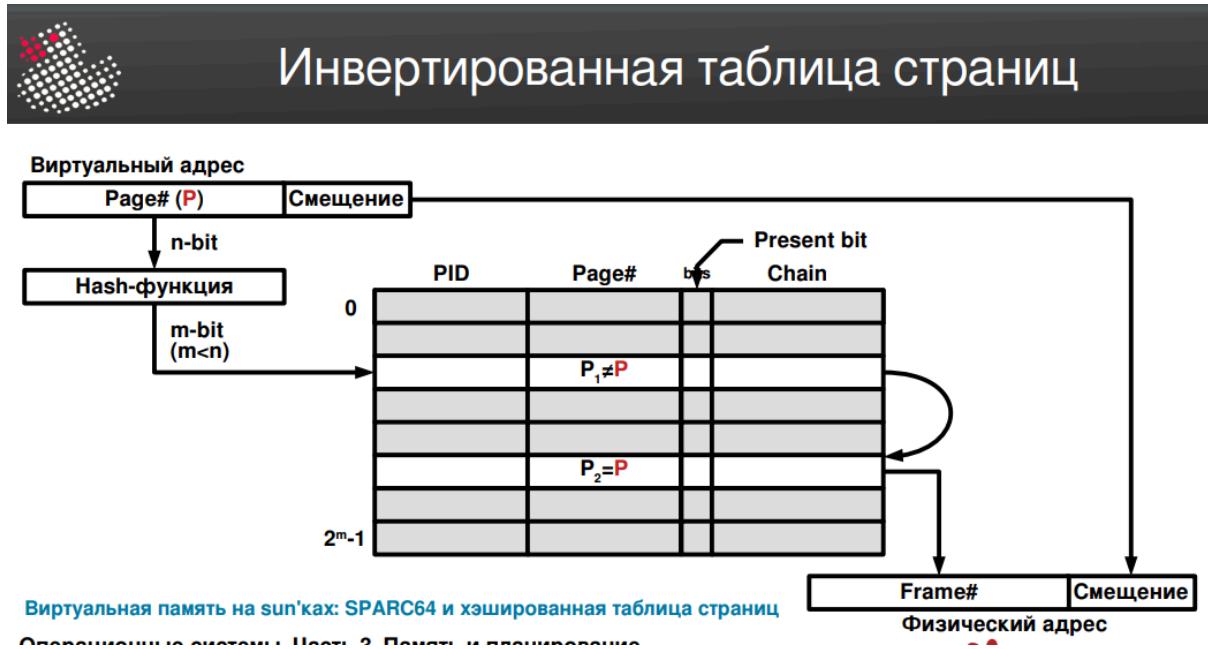
Мы берем из виртуального адреса PDE и PTE, и кидаем их в TLB, если там есть запись, то мы сразу получаем номер кадра в физической памяти, добавляем к нему наше смещение и получаем физический адрес.

Если нашей страницы нет в TLB, то мы берем наш регистр CR3, суммируем его с PDE, находим этот индекс в таблице PDT, тем самым получаем нужную нам таблицу страниц, далее прибавляем к ней наш PTE, получаем нужную страницу, далее прибавляем к ней смещение и получаем физический адрес.

Однако при полной трансляции адреса стоит помнить что на этапах получения таблицы страниц (major fault - в списке сегментов есть, но нет в таблице PDT/PT) и конкретной страницы (minor fault) может возникнуть ситуация, что данных записей нет в первичной памяти, тогда нам необходимо сначала подгрузить нужные страницы, и потом продолжить работу.

(При TLB miss) После получения физического адреса появится новая запись в TLB и будет вытеснена старая.

45. Инвертированная таблица страниц.



Кто она такая и кто ее инвертировал - загадка; Понятнее будет сказать **хешированная таблица страниц**. Выполняет абсолютно ту же задачу, что и TLB - преобразует виртуальный адрес в физический. Реализована так: адрес виртуальной страницы скармливается хэш-функции и получаем индекс в хэш-таблице (посередине на слайде), в которой по индексу и находим нужный физический адрес, если он выше размаплен. Фишка в том, что реализована через цепочки (для случаев коллизии, когда у двух вирт. адресов одинаковый хэш), то есть каждая ячейка таблицы может содержать ссылку на другую ячейку с тем же хешом.

Преимущества инвертированной таблицы:

- в сокращении таблицы маппингов. (я так понял просто за счет хеша вместо тупой таблички).
- поиск по хешу довольно быстрый, хотя ассоциативная TLB тоже даст просраться.
- когда мало размапленных страниц, двухуровневая организация тратит кучу памяти на размещение PDT и PT (типа для всего одной записи понадобится 8кб), а тут у нас просто лаконичный массивчик, который занимает столько, сколько нужно

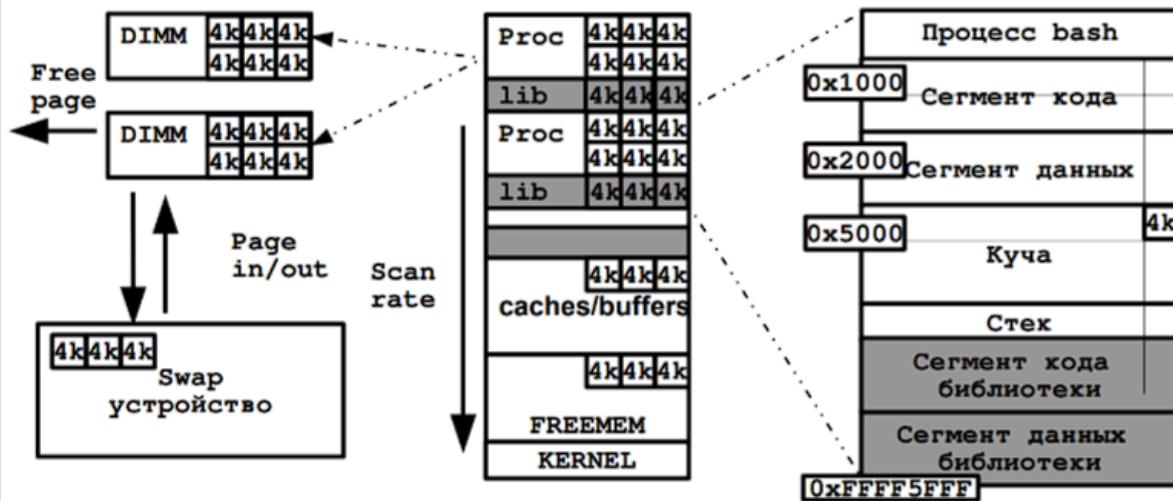
46. Сегментно-страничная виртуальная память.

hackmd.io/@come-ill-foo/B1_DBATrY

Мб пригодится (а мб и нет)



Сегментно-страничная виртуальная память



8.2. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ОПЕРАЦИОННОЙ СИСТЕМЫ

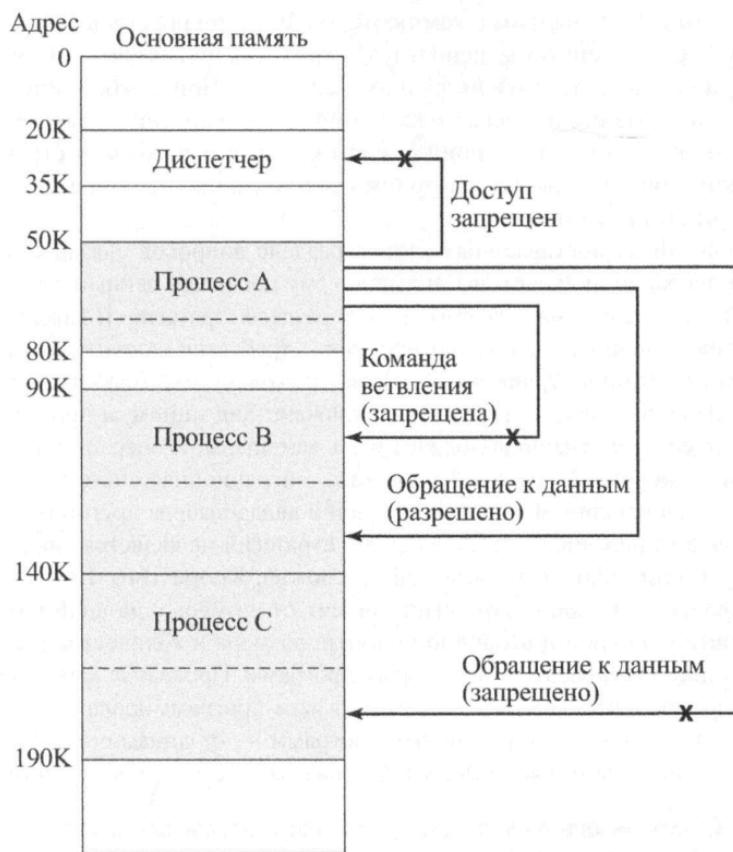


Рис. 8.13. Отношения защиты между сегментами

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

Послушать Оригинал можно тут:

▶ ОС #3-2. Виртуальная память (47:47 до 1:02:22)



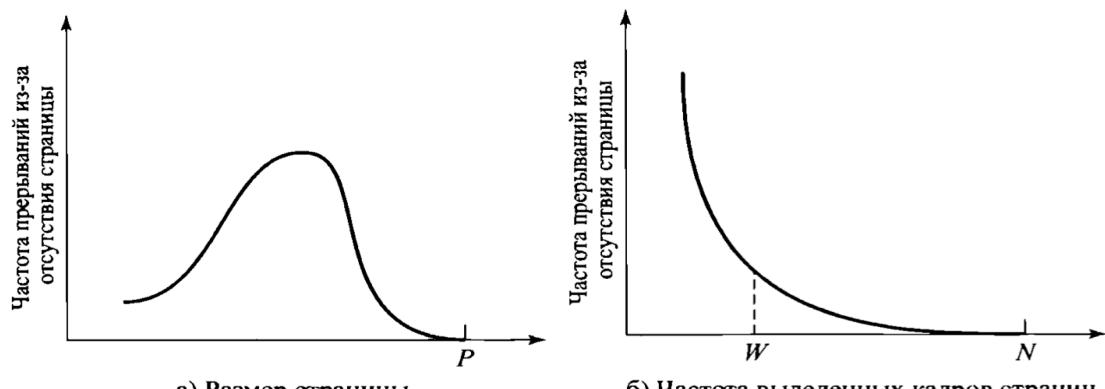
Размер страницы

- При разработке ОС необходимо определится с размером страницы.
Важно учитывать:
 - Размеры таблиц страниц
 - Внутреннюю фрагментация
 - Количество page-fault при трансляции адреса
 - Скорость взаимодействия со вторичной памятью (и размер блока)
 - Локальность данных (в многопоточных приложениях ниже)
 - Количество промахов TLB, размер TLB, размер кэша L1, L2, L3, и др.
- Современные процессоры могут работать с разными размерами страниц (large, huge pages)
 - Intel 4Кб, 2Мб, 1Гб
 - Sparc64 VI 8Кб, 64Кб, 512Кб, 4Мб, 32Мб, 256Мб

С одной стороны: внутренняя фрагментация находится в прямой зависимости от размера страницы. Чем размер меньше - тем меньше внутренняя фрагментация.

С другой стороны: Чем меньше страницы, тем больше их требуется для процесса, тем самым мы повышаем размеры и нагрузку на TLB. Также при работе с большими программами мы можем получать двойное прерывание из-за отсутствия страницы в памяти(->таблица страниц->страница)

Также на размер страницы влияет вторичная память. В чем суть? На дорожке располагается несколько блоков, и проще сделать размер страницы таким, чтобы уместить на него цепочку блоков на дорожке, а не один блок дорожки, чтобы избежать сильного вращения шпинделя диска.



P — размер процесса
 W — размер рабочего множества
 N — общее количество страниц процесса

На частоту промахов сильно влияет локальность данных. В стандартном однопоточном приложении мы работаем с одним фрагментом памяти. Треды же будут работать с данными которые будут находиться в их локальных частях.



Стратегии ОС по работе с виртуальной памятью

- Стратегия выборки страниц из вторичной памяти:
 - По требованию, предварительная выборка
- Стратегия размещения
 - В современных NUMA системах скорость выше рядом с процессором процесса/потока
- Стратегия очистки (выгрузки во вторичную память)
 - По требованию и предварительная очистка
- Управление многозадачностью
 - Выгрузка процессов целиком
- Стратегии замещения (см. далее)
- Управление резидентной частью процессов (см. далее)

Операционные системы. Часть 3. Память и планирование
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Выборка по требованию:

Обратились к странице и подгрузили ее

Предварительная выборка:

Обратились к данным и понимаем что данные из того же самого сегмента лежат последовательно дальше, тоже подгружаем

Очистка:

Смысл в том, что если какая-то страница долго не используется, ее выгружают или очищают в зависимости от контекста. Кусок кода бессмысленно выгружать, к примеру.

Многозадачность:

Ограничение кол-ва процессов которые запускаются на процессоре(разумеется не одновременно)

48. Стратегии замещения страниц ОС. Часовой Алгоритм.

Управление резидентной частью процесса.

Послушать Оригинал можно тут:

<https://youtu.be/pB7cPle-o24?t=3737> (1:02:17 до 1:21:11)

Нам нужно понимать, какие страницы и с какими приоритетами загружать в память и выгружать из неё, причём желательно уметь это предсказывать. Для этого можно учитывать прошлое поведение.

Некоторые фреймы выгружать вовсе нельзя (locked frames): части ядра, буферы. А некоторые кадры используются настолько часто, что их выгрузка повлияет на кучу процессов (н-р код разделяемой библиотеки)



Стратегии замещения ОС

- 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



Стратегии замещения ОС (2)

- «**Оптимальная стратегия**» - страница к которой обращение будет через наибольший промежуток времени
 - Мы не Пифии, не Оракулы мы.
 - Обычно используется для сравнения с остальными
- **LRU (least recently used)** — заменять наименее используемые страницы
 - Сложность определения наименее используемых
- **FIFO** — просто реализовать, но неэффективно
- **Clock algorithm**

“Оптимальная стратегия” — невозможная стратегия, используемая для сравнения других стратегий. Такую можно построить только зная наперёд все запросы к памяти. Почему невозможно даже на понятных процессах: процессы зависят от входных данных: поменяются данные — поменяются и обращения к памяти

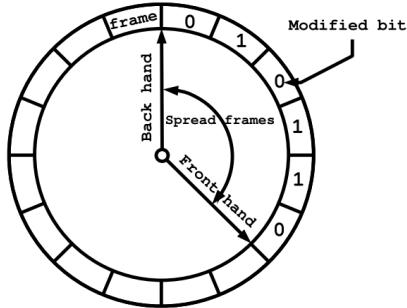
Замещение страниц: LRU (Least Recently Used)

Основывается на том, когда страница была использована последний раз. Если нам нужно додгрузить новую страницу в память, то мы вытесняем ту, что использовалась давнее всего. Наименее используемые определять == нагрузка на систему в момент свопинга, поэтому в чистом виде используется редко

Замещение страниц: FIFO (First In First Out)

Реализовывается просто: первая загруженная страница будет выгружена первой. Не используется в чистом виде из-за неэффективности.

Замещение страниц: Часовой алгоритм



Каждой странице памяти назначаем бит изменения, если мы обращаемся к странице памяти, то ставим этот бит. По всем страницам памяти ходят две “стрелки” (front и back hand, FH и BH), которые врачаются одновременно с одной скоростью. FH сбрасывает бит изменения, BH проверяет этот бит и помещает страницы, у которых он не установлен, в буфер вытеснения. Расстояние между стрелками выставляется динамически, чем оно меньше, тем быстрее страницы будут вытесняться.

Этот алгоритм в современных системах занимает слишком много времени, поэтому с него системы перешли к другим, более сложным алгоритмам.

Управление резидентной частью процесса (с 1:16:11 в лекции)

Резидентная часть процесса — определения он не дал, но это та часть, которая выделена процессу в памяти. Её размером управляет система, при этом она учитывает:

- Нет смысла держать весь процесс в памяти целиком
- Чем меньше процессов в памяти, тем больше их можно загрузить в память
- Если мало давать памяти процессу, то будет много pagefault-ов
- После N-страниц дополнительное выделение памяти процессу не даёт кардинального снижения pagefault-ов



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

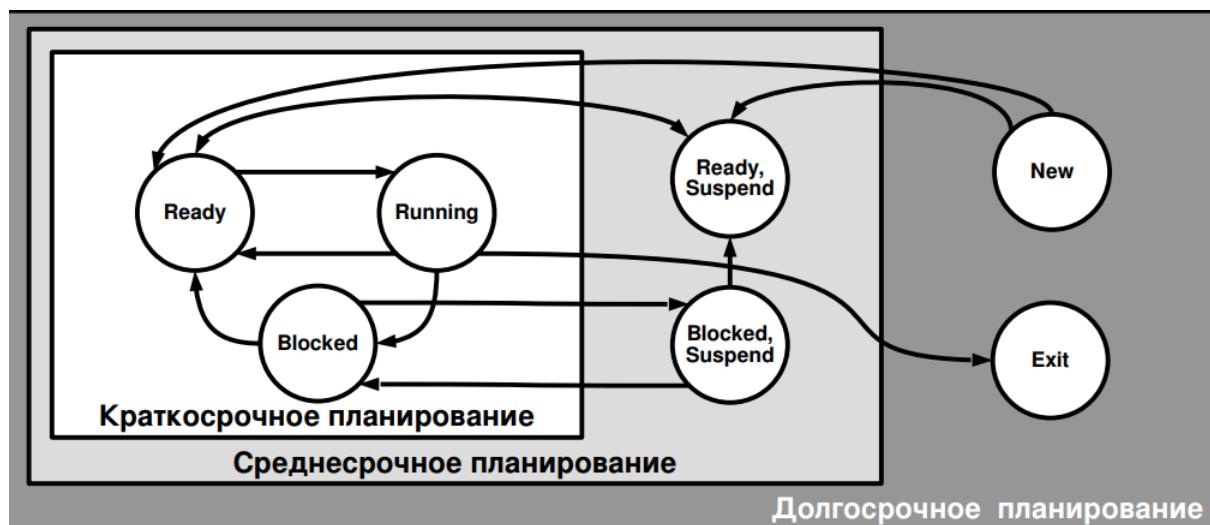
Таблица 8.5. УПРАВЛЕНИЕ РЕЗИДЕНТНЫМ МНОЖЕСТВОМ

	Локальное замещение	Глобальное замещение
Фиксированное распределение	<ul style="list-style-type: none"> Количество кадров процесса фиксировано Страница для замещения выбирается среди выделенных процессу кадров 	<ul style="list-style-type: none"> Невозможно
Переменное распределение	<ul style="list-style-type: none"> Количество выделенных процессу кадров может время от времени изменяться Страница для замещения выбирается среди выделенных процессу кадров 	<ul style="list-style-type: none"> 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.

Виды планирования процессов:

- 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)



Критерии краткосрочного планирования:

- Пользовательские, связанные с производительностью
 - Turnaround time - время оборота
 - Response time - время отклика
 - Deadline - предельное время
- Пользовательские, иные
 - Predictability - предсказуемость (независимость от остальной загрузки системы)
- Системные, связанные с производительностью
 - Throughput - пропускная способность
 - Processor Utilization - загруженность процессора
- Системные, иные
 - Fairness - справедливость
 - Enforcing priorities - принудительная приоритизация
 - Balancing resources - сбалансированная загрузка

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

а

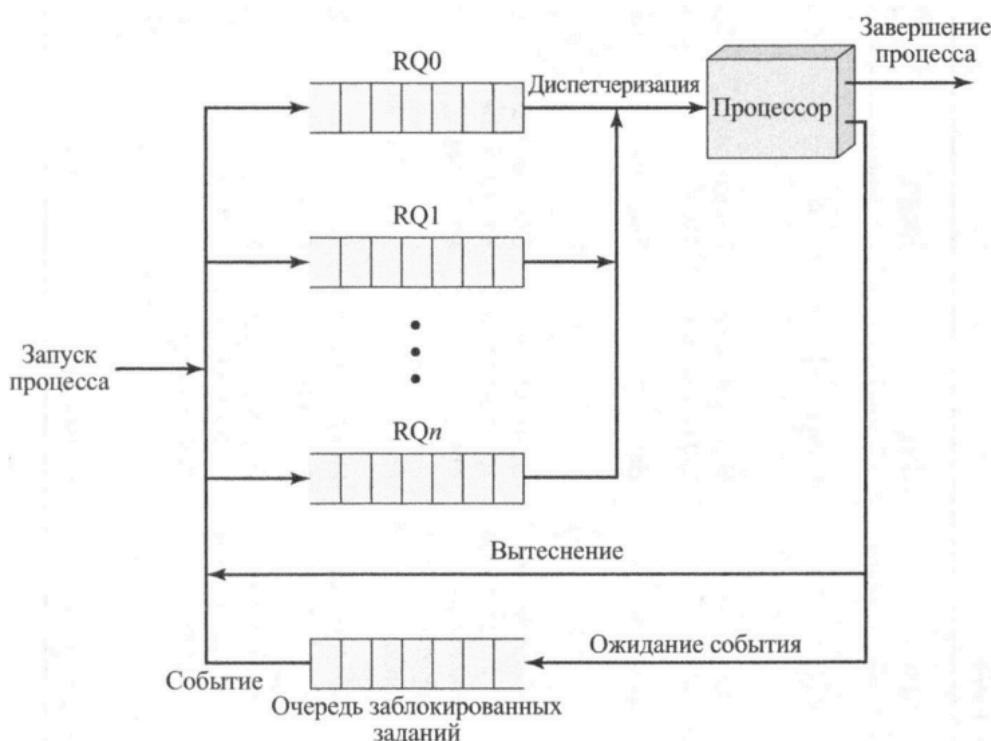


Рис. 9.4. Планирование с учетом приоритета процессов

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).

В ОС процессам назначаются приоритеты - числа, позволяющие планировщику определить, каким процессам отдавать предпочтение.

Unix - обратная зависимость (больше число - ниже приоритет), Windows - прямая.

Процессы ставятся в очереди и выбираются из этих очередей. Естественно, в первую очередь по умолчанию выбирается процесс с наивысшим приоритетом.

В системах с вытесняющей многозадачностью это тоже работает (по окончании проц. времени процесс заменяется на более высокоприоритетный).

Внутри ядра существует столько очередей, сколько приоритетов.

Когда в очереди находится несколько процессов, для выбора процесса используются дополнительные стратегии, например, FCFS или RR.

Процесс с низким приоритетом может испытывать недостаток проц. времени (голодание). Для недопущения этого в современных ОС приоритеты могут динамически изменяться.

Проблема инверсии приоритетов: [56 билет](#)

Через shell приоритет можно менять командой nice.

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

Стратегия планирования — алгоритм выбора следующего исполняемого процесса из пула готовых к исполнению

1. First Come First Served (FCFS): пул готовых потоков - FIFO очередь. Мы выбираем тот процесс, который раньше всех пришёл в очередь исполнения. Может комбинироваться с использованием приоритетов. Тогда для каждого приоритета у нас будет своя очередь и сначала будем выбирать из очередей с более высоким приоритетом
2. Round Robin (RR): очередь процессов замыкается и становится круговой, используется квантование времени
3. Shortest Process Next (SPN): берём процесс с минимальным ожидаемым временем исполнения (точно его определить невозможно)
4. Shortest Remaining Time (SRT): берём процесс с наименьшим ожидаемым временем до завершения (разность ожидаемого времени исполнения и того, сколько уже исполнялся)
5. Highest Response Ratio Next (HRRN)
6. Feedback: динамически меняем приоритет. Если процесс долго находится в Wait, повышаем, если много ест процессорного времени, понижаем



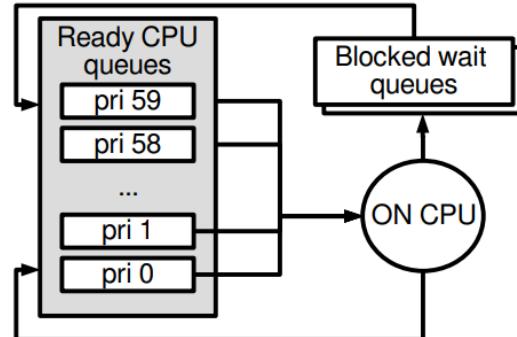
Сравнение стратегий

	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
Throughput	-	Can be low if TQ low	High	High	High	-
Response 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 prefer 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 е

52. Feedback планировщик и классы планирования ОС UNIX SVR4.

globpri	quantum	tqexp	slpreat	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



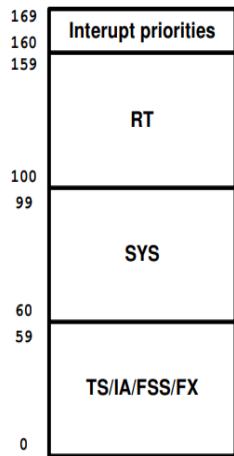
globpri - глобальный приоритет,
quantum - выделяемый квант времени (мс),
tqexp - приоритет, если квант полностью выбран CPU,
slpreat - приоритет, если ожидаем ресурса
maxwait — граничное время длинного ожидания (мс)
lwait - приоритет, в случае длинного ожидания ресурса

Про классы планирования:

- TS - дефолтный feedback (для обычных пользовательских приложений)
- IA - +n к приоритету (типа для графических оболочек и прочего)
- FIXED - неизменяемый приоритет
- System - выше пользовательских
- RT - задачи реального времени, которые должны выполняться в первую очередь
- Interrupt - прерывание
- Fair Share schedule - всем процессам дается какое-то числовое значение и каждый процесс получает в среднем такую долю процессора, которая

совпадает с долей его числа в сумме всех чисел процессов

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 аффинити для NUMA

Если у нас нет никаких указаний об относительной продолжительности процессов, то мы не можем использовать ни одну из рассмотренных стратегий, SPN, SRT или HRRN. Еще один путь предоставления преимущества коротким процессам состоит в применении штрафных санкций к долго выполняющимся процессам. Другими словами, раз уж мы не можем работать с оставшимся временем выполнения, мы будем работать с затраченным до настоящего момента временем.

Вот как этого можно достичь. Выполняется вытесняющее (по квантам времени) планирование с использованием динамического механизма. Процесс при входе в систему помещается в очередь RQ0 (см. рис. 9.4). После первого вытеснения и возвращения в состояние готовности процесс помещается в очередь RQ1. В дальнейшем при каждом

вытеснении этот процесс вносится в очередь со все меньшим и меньшим приоритетом. Соответственно, быстро выполняющиеся короткие процессы не могут далеко зайти в иерархии приоритетов, в то время как длинные процессы постепенно теряют свой приоритет. Таким образом, новые короткие процессы получают преимущество в выполнении перед старыми длинными процессами. В рамках каждой очереди для выбора процесса используется стратегия FCFS. По достижении очереди с наиболее низким приоритетом процесс уже не покидает ее, всякий раз после вытеснения попадая в нее вновь (таким образом, эта очередь, по сути, обрабатывается с использованием кругового планирования).

На рис. 9.10 проиллюстрирован этот механизм планирования; пунктирной линией показан путь длинного процесса по различным очередям. Такой подход известен как **многоуровневый возврат** (multilevel feedback), поскольку при блокировании или вытеснении процесса осуществляется его возврат на очередной уровень приоритетности.

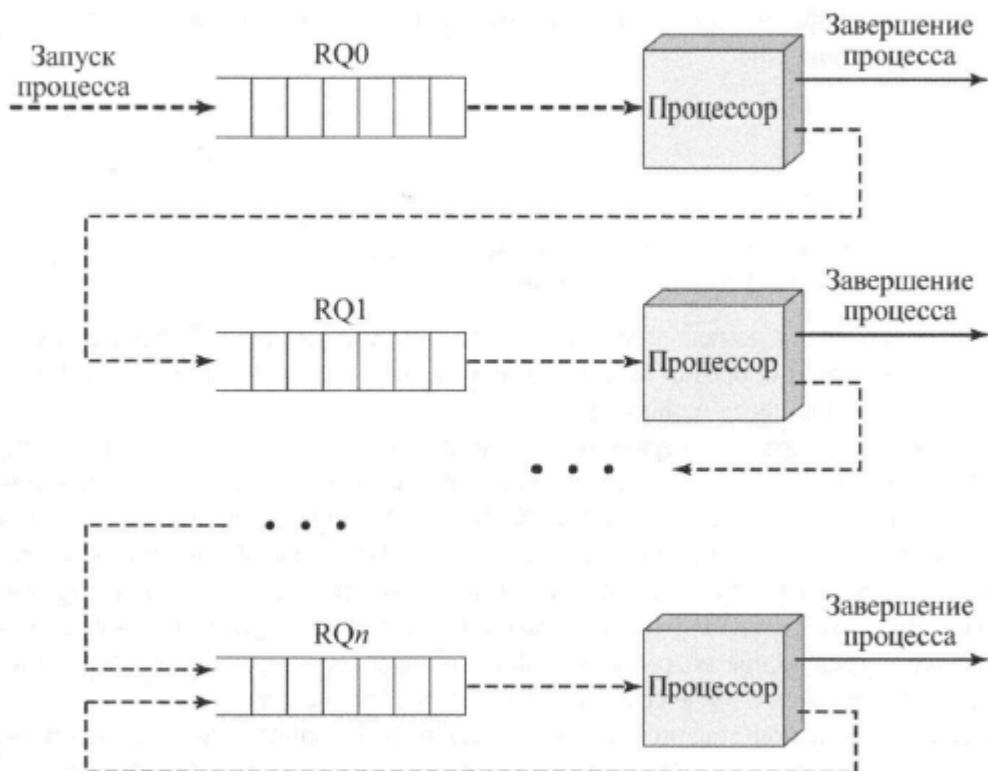


Рис. 9.10. Планирование со снижением приоритета

Имеется несколько разновидностей данной схемы. В простейшем случае вытеснение выполняется так же, как и в случае применения стратегии кругового планирования, — через периодические интервалы времени. В нашем примере (рис. 9.5 и табл. 9.5) использован именно этот метод, с квантом времени, равным 1.

При такой простой схеме имеется один недостаток, заключающийся в том, что время оборота длинных процессов резко растягивается. В системе с частым запуском новых процессов в связи с этим вполне вероятно появление голодания. Для уменьшения отрицательного эффекта мы можем использовать разное время вытеснения для процессов из разных очередей: процесс из очереди RQ0 выполняется в течение одной единицы времени и вытесняется; процесс из очереди RQ1 выполняется в течение двух единиц времени

и т.д. — процесс из очереди RQ_i выполняется до вытеснения в течение 2^i единиц времени. Использование этой схемы также проиллюстрировано на рис. 9.5 и в табл. 9.5.

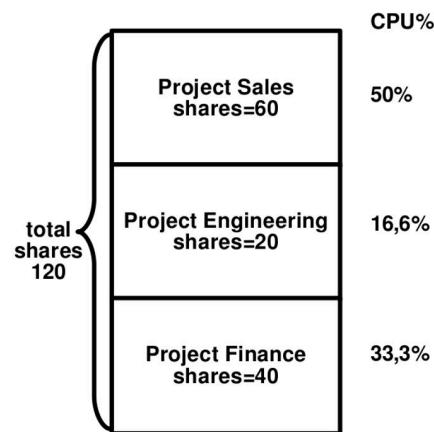
Даже при выделении процессу с более низким приоритетом большего количества времени для выполнения не удается полностью избежать голодания. Еще одним средством против голодания может служить перемещение процесса в очередь с более высоким приоритетом, если процесс не был обслужен в течение некоторого порогового времени для данной очереди.

53. Справедливое планирование.



Справедливое планирование SVR4

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



Справедливость (Fairness) - всем процессам из одного класса, претендующим на ресурс, предоставляется примерно одинаковый доступ.

Используется по отношению к группам процессов или пользователей.

При справедливом планировании каждой группе процессов выделяются кванты администрации (shares). Доля ресурсов группы вычисляется как отношение ее shares к их сумме.

Пример со слайда: три группы - 60-20-40, сумма - 120, доля ресурсов - $60/120 = 50\%$, $20/120 = 16,6\%$. $40/120 = 33,3\%$.

В Solaris используется в виде замены TS/IA классов.

В Solaris этот планировщик использовал “плавающую точку” внутри ядра, поэтому на некоторых моделях процессорах его не рекомендовалось использовать.

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



Типы 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



Synchronization Granularity

- Гранулярность синхронизации — 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 (грубая) — 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 простой 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.



ОС реального времени

- Hard & Soft realtime, Периодичность работы
- Основные Требования
 - Determinism — выполнение операций в предопределенный интервал времени. Мера: время от прерывания до начала его обработки.
 - Responsiveness — сколько времени требуется для ответа?
 - User control — управление планированием со стороны пользователя
 - Reliability — повышенные требования по сравнению с «обычными» ОС
 - Fail-soft operation — мягкая реакция на ошибки (не kernel dump)
- Планировщик: строгое использование приоритетов с вытеснением и ограниченные, минимальные задержки



Deadline планирование

- Важны 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 Scheduling — см. Столлинг гл. 10.2

Насчет rate monotonic scheduling - не вижу смысла вставлять скринь из книги. Если

кто-то считает иначе, вставьте. Там 3 странички, достаточно просто прочитать.

Ну ты и долбаеб

- **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.

10.2. ПЛАНИРОВАНИЕ РЕАЛЬНОГО ВРЕМЕНИ 561

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.

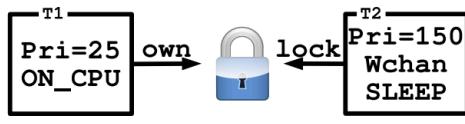
- **Динамическое планирование наилучшего результата.** При этом анализ осуществимости планирования не выполняется; система пытается удовлетворить все предельные сроки и снимает те выполняющиеся процессы, предельные сроки которых нарушены.

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

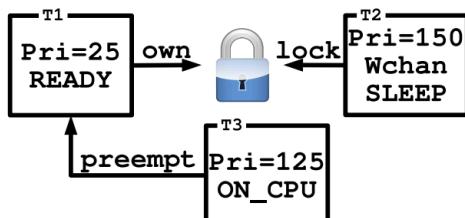


Инверсия приоритетов

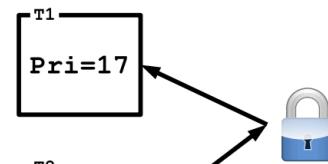
- Решается при помощи наследования приоритетов



a) Bounded priority inversion



b) Unbounded priority inversion



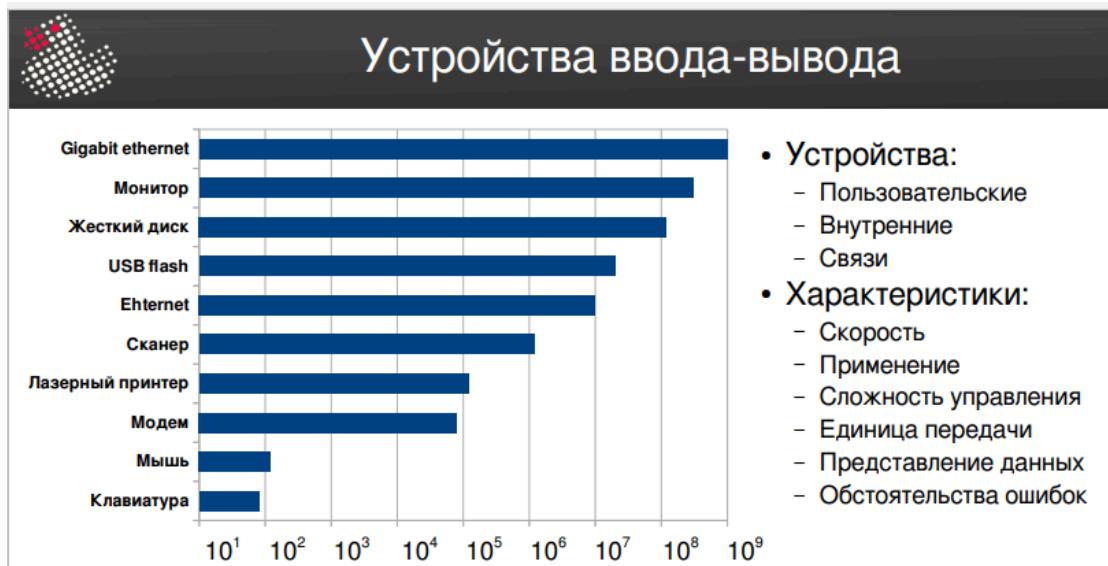
c) Blocking chain

Есть несколько типов инверсий приоритетов

- Ограниченнная - у нас есть процесс с высоким приоритетом, и ему нужен ресурс который занят процессом с более низким приоритетом. То по хорошему мы должны дождаться завершения работы процесса, и по окончании захватить ресурс.
- Неограниченный - приоритетные потоки вытесняют менее приоритетные. Для решения этой проблемы придумано наследование приоритета, т.е. поток, которому нужен ресурс дает свой приоритет потоку на ресурсе, и другим потокам сложнее его захватить.
- Цепочка блокировок - передача уровня приоритета всей цепочке потоков, с запросом на ресурс

Если обобщить - 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 высоко приоритетное задание блокируется в ожидании ресурса; оно должно заканчиваться, когда ресурс освобождается задачей с более низким приоритетом.

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



Как видно на картинке, разные устройства ввода-вывода на целые порядки отличаются друг от друга по скорости передачи данных. На данный момент средняя скорость информационного обмена устройств ввода-вывода колеблется от 10^1 до 10^9 бод (байт в секунду). В связи с этим операционная система должна эффективно использовать как быстрые, так и медленные устройства (не замедляя работу друг друга).

Все устройства, подключаемые к компьютеру можно разделить на

- пользовательские, т. е. те, с которыми взаимодействует пользователь, и у них обычно похожие друг на друга характеристики;
- внутренние, т. е. находящиеся внутри вычислительной машины. В основном эти устройства подключаются драйверами, которые поставляются или как в случае Windows в комплекте с операционной системой, или как в Linux, где они вставлены в ядро. И очень важно, чтобы операционная система имела полный набор необходимых драйверов, т. к. в противном случае «заставить» работать устройство ввода-вывода без драйвера не получится;
- устройства связи (модемы, устройства Ethernet и т. п.). Следует заметить, что скорости информационного обмена эти устройств могут существенно различаться;

У устройств ввода-вывода есть различные характеристики, зависящие от их применения. В том числе:

- скорость (и это, как правило, основная характеристика);
- особенности использования;
- сложность управления;
- единица передачи информации устройством (существуют блочная и потоковая символьная передача);
- представление данных;
- обработка ошибок (включающая определение обстоятельств возникновения и особенностей обработки).

Приведенное разнообразие устройств и их характеристик означает, что операционная система должна внутри себя содержать такую инфраструктуру, которая обеспечивает совместную работу разнородным устройствам внутри вычислительной системы.



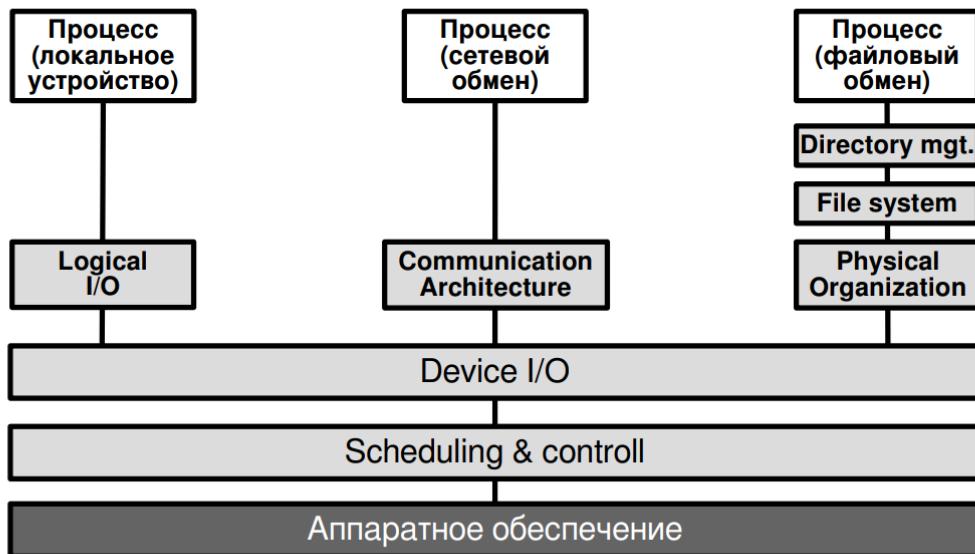
Развитие ввода вывода

- 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)

Тенденция развития одна - отстранить процессор от затратной работы ввода-вывода и делегировать эту работу чему-нибудь другому.



Логическая структура ввода-вывода

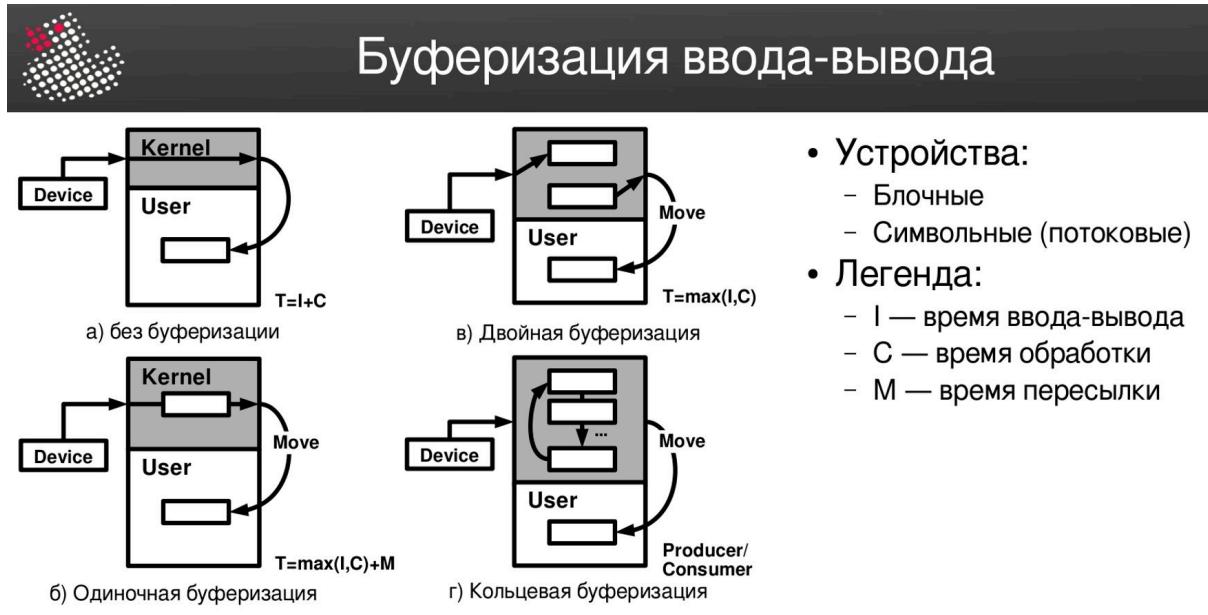


- **Логический ввод-вывод.** Модуль логического ввода-вывода обращается с устройством как с логическим ресурсом и не обращает внимания на детали фактического управления устройством. Логический модуль ввода-вывода работает посредником между пользовательскими процессами и устройством, позволяя им работать 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.

Обобщая картинку, можно сказать так: при запросе на ввод-вывод нам в первую очередь необходимо идентифицировать место, куда совершается операция. Это может быть локальное устройство, узел в сети, или файл в системе. Для каждого из этих вариантов необходимо провести первоначальную подготовку к обмену: настроить интерфейс обмена, определить адрес получателя (номер порта, дескриптор файла и т.п.), найти драйвер устройства и т.д. Далее идут стадии “Устройство ввода-вывода” и “Планирование и контроль” (описаны выше), а затем данные поступают в аппаратуру, причем с развитием аппаратуры важно следить, чтобы не нарушались никакие интерфейсы и порты для передачи.

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

Билет не написан, пусть пока будет так.



Как было показано ранее - типы устройств бывают разными. При этом операционная система может считывать всегда только ограниченное число информации с устройства ввода-вывода напрямую. Например, из контроллера «сериального порта» можно прочитать одновременно всего 1 байт, который туда приходит. И, прочитав его, необходимо решить, что с этим байтом надо сделать.

В этом случае у операционных систем есть два выбора: буферизировать полученное (т. е. сложить все в промежуточный буфер) или не буферизировать. Все операционные системы по-разному работают с символьными и с блочными устройствами.

Блочные устройства работают, передавая блоки данных. Характерный пример: жесткий диск. Минимальная единица информации, которая может оттуда прийти будет размером с сектор жесткого диска, что в современном мире составляет 4К.

Внешний обмен с жестким диском выглядит так: посыпается команда на обмен и устанавливается адрес по DMA, куда контроллер должен передать информацию с диска. После этого, в момент, когда диск прочитает этот блок, вызовется прерывание. Потом внутри этого прерывания происходит настройка на передачу блока в оперативную память.

Для блочных устройств необходимо решить, надо ли буферизировать данные, которые приходят, т. е. стоит ли класть ли их в промежуточный буфер, и зачем это надо делать.

Схема, представленная на рис. а) «без буферизации» - подразумевает, что пользовательская программа запросила у ядра какой-нибудь фрагмент файла: ядро превратило этот фрагмент файла в блок с каким-то номером на жестком диске; после этого операционная система запросила эту информацию у драйвера устройства; драйвер устройства ее подготовил; и после этого непосредственно при помощи DMA (как показано стрелкой) в пользовательское пространство записался блок данных.

Вопрос: отсутствие буферизации в устройствах ввода-вывода хорошо это или плохо? Естественно, если такой режим есть – значит в нем есть необходимость.

На практике режим «без буферизации» (который иногда называют Direct I/O) максимально быстро отдает данные приложению. Но у этого режима есть одна проблема. Проблема заключается в том, что в системе применяется вытесняющая многозадачность, при которой процессы иногда могут уходить в своп: дали команду; пользовательский процесс ушел в своп; при этом устройство ввода-вывода стало готовым к приему-передаче через некоторое количество времени и пытается вернуть запрошенные данные обратно, а страницы, куда отдавал данные – нет (она в свопе).

Это означает, что операционной системе надо быть предельно внимательной при этом режиме. В частности, ОС может «започить» страницу в памяти, если известно, что в нее должен прийти ответ от устройства. И этим не давать процессу высвободиться. Но это тоже не очень хороший вариант, поскольку, когда таких процессов в системе много, то свопинг перестает работать.

Поэтому прямым (безбуферным) режимом если и пользуются, то только для определенных случаев. Например, прямой режим передачи данных в пользовательское адресное пространство широко распространен в базах данных. Например, у Oracle его кэш целиком лежит в SGA-области (system global area) и эту область можно подключить к «сырым устройствам» без буферизации.

Чтобы страницу не «започить» (как это необходимо в передаче «без буфера»), была предложена одиночная буферизация, представленная на рис. б).

В этом случае в ядре создается специальная область для ввода-вывода; которая запрашивается для процесса ввода-вывода. А после того, как ввод-вывод до конца закончится, данные, которые накоплены в ядре с помощью операции Move переносятся в пользовательское адресное пространство. Причем надо понимать, что операция Move по сравнению с самим вводом-выводом, не очень длительная операция.

Время всей операции ввода-вывода состоит из трех частей: I – время ввода-вывода, С – время обработки пользовательским процессом, M – время пересылки.

Для безбуферного обмена общее время работы программы время получается $T=I+C$ – и оно будет самым большим из всех возможных случаев передачи.

Для однобуферного обмена время операции ввода-вывода сокращается, поскольку пользователь в то время, пока читаются в буфер следующие блоки, может обрабатывать данные, которые ему уже пришли. В этом случае грубая оценка общего времени работы программы покажет, что общая эффективность работы программы будет состоять из максимального времени либо обмена, либо вычислений, плюс время пересылки (Move): $T=\max(I,C)+M$. Единственное, что надо при этом иметь в виду – это то, что пока буфер переносится операцией Move, устройство ничего не может делать.

Следующим более продвинутым этапом развития режимов передачи данных в процессе ввода-вывода стала двойная буферизация, представленная на рис. в).

В этом случае в ядре существует два буфера. С одним из буферов работает пользовательская программа, и из него переносится информация в пользовательское адресное пространство с использованием Move. А в это время во второй буфер (если имеется много последовательных данных) записываются следующие данные. Это позволяет общее время работы программы еще более сократить до величины $T=\max(I,C)$ (максимальной величины либо ввода-вывода, либо вычислений). Но при этом растет требование к памяти, которая у нас есть на стороне ядра.

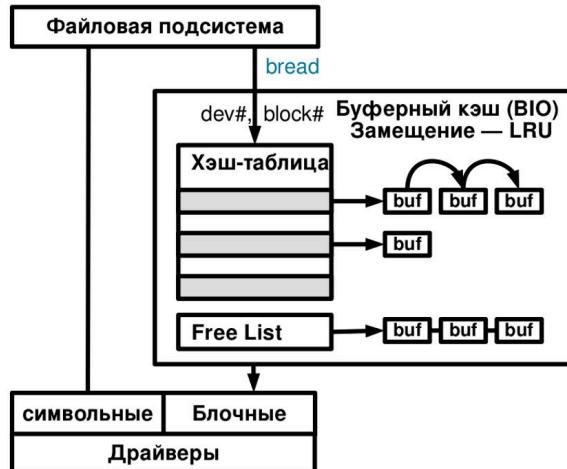
Кроме всего прочего, существует еще и кольцевая буферизация (представленная на рис. г).

Некоторые аналоги этого режима сейчас широко используются в операционных системах.

Для организации такого вида ввода-вывода используется паттерн, связанный с примитивами синхронизации, который называется Producer/Consumer. В этом случае устройство выступает как Producer, а программа для чтения выступает как Consumer.



Ввод-вывод в SVR4



- Два вида:

- Буферизированный ввод-вывод (через bio)
- Небуферизированный ввод-вывод (через DMA)

- Для медленных устройств (терминалы...)

- Используются отдельные символьные буфера

Все запросы, которые генерятся в результате вычислений, идут или из файловой системы, или из процессов, которые непосредственно обращаются к устройствам.

Если запросы идут из файловой системы, то данные обычно буферизируются.

Существуют два типа устройств: символьные и блочные.

В символьных устройствах существуют свои минимальные по размерам промежуточные буфера, из которых данные отправляются непосредственно в контроллер.

Для дисков также существуют символьные устройства (иногда их называют «сырые устройства»). И разница между ними такая, что в данном случае происходит так называемая «безбуферизация» - то есть буфера нет, и данные приходят непосредственно в сам процесс. При этом операционная система «ложит» те страницы, в которые необходимо переносить информацию, что может привести к «не выгрузке» процесса из памяти.

Если используются блочные (или – обычные) устройства, то внутри UNIX используется буферный кэш.

В UNIX существует отдельная подсистема, называемая BIO (buffered input/output), у которой есть специальные вызовы bread, write, которые связаны с работой с буферами обмена с дисковым устройством.

При этом существует набор свободных буферов; и есть отдельный указатель на них - Free List, содержащий список буферов, которые уже созданы в программе и могут быть использованы. Соответствующая переменная ядра указывает, какой объем можно от оперативной памяти занять под использование буферов устройств.

При этом каждый запрос, который генерится в системе, пройдет через файловую подсистему и вызовет чтение из файла, после чего запрос превратится в запрос буфера bread.

Для быстроты поиска в буфере в оперативной памяти используется хеш-таблица, ключом в которой являются dev# (номер устройства) и block# (номер блока на этом устройстве). Следующим шагом в хештаблице происходит поиск соответствующей цепочки, у которой в «чейне» расположены ссылки на те буфера, которые есть в памяти.

Если буфера нет, то система прочитает его с диска и добавит его в список.

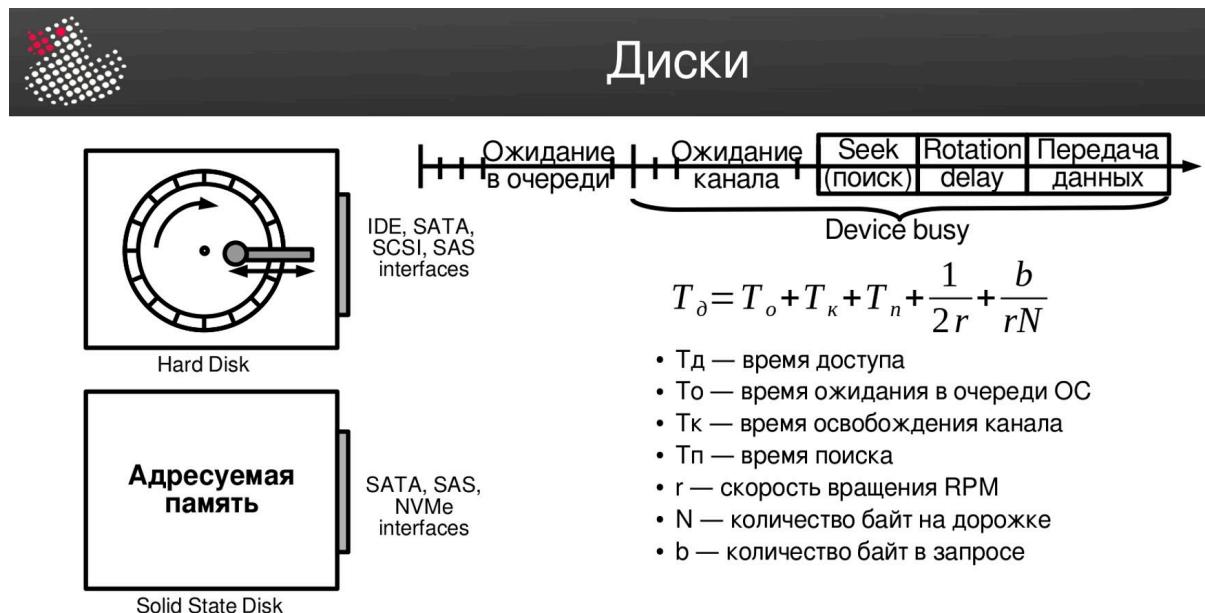
Когда памяти перестает хватать, используется алгоритм LRU, который проверяет, какие буфера могут устареть, после чего они помещаются в Free List, из которого потом забираются в хеш-таблицу.

Таким образом, весь блочный вывод состоит из двух частей:

- буферизированный ввод-вывод (с использованием bio),
- небуферизируемый ввод-вывод (через DMA).

Для медленных устройств нет смысла выделять большие буфера данных. Поэтому для них в драйвере используются отдельные символьные буфера, куда помещается информация, чтение которой производится.

59. Диски и дисковое планирование.



Операционные системы. Часть 3. Память и планирование

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

В настоящее время устройства хранения, называемые дисками, бывают разными. На слайде представлены два типа дисков.

Первый тип – это обычный жесткий диск. У него внутри есть несколько так называемых «блинов», которые вращаются с постоянной скоростью. Считывающая головка расположена между дорожками. Дорожка разбита на сектора. Такой жесткий диск (HDD) может иметь различные интерфейсы. В том числе: IDE (старый и уже не применяется), SATA (один из наиболее популярных сегодня), SCSI (раньше существовал такой полноформатный интерфейс, но из SCSI сегодня используется только протокол команд, которыми обмениваются с дисковыми устройствами), SAS (этот интерфейс сегодня имеют все диски, раньше работавшие как SCSI).

Сегодня все чаще используются «диски» другого типа – Solid State Disk. Они не содержат никаких движущихся частей. Внутри них в микросхемах находится адресуемая память. При использовании таких устройств не надо ждать время оборота диска и время перемещения головки до необходимого сектора. А данные можно брать прямо из самого «диска». Такие диски сегодня также выпускаются в вариантах SATA и SAS, но есть также и более производительный и часто использующийся сегодня в персональных компьютерах интерфейс - NVMe.

Сравним эффективность этих двух видов дисков.

Жесткие диски (HDD) лучше справляются с последовательным чтением. То есть таким случаем, когда блоки файла расположены последовательно, и за каждый оборот диска происходит чтение данных из нескольких последовательных блоков. Если же мы начинаем «случайно» обращаться в различные места этого диска, то начинают происходить процессы, сильно снижающие эффективность таких устройств. Они состоят в том, что головка каждый раз перемещается от сектора к сектору, и каждый раз приходится ожидать окончания его поворота и передачи данных. Поэтому скорость при случайном доступе к таким дискам очень падает.

У SSD такой проблемы нет, т. к. это просто память, не имеющая подвижных частей. И процесс ее чтения практически не зависит от того, производится последовательное или случайное чтение. Вернее, задержки будут, но не такие, как в случае HDD, когда скорость обмена при случайных обращениях может упасть «на порядок» по сравнению с последовательным чтением.

Общий доступ к дисковым устройствам делится на несколько фаз.

В начале существует ожидание в очереди, которое происходит на стороне операционной системы, в связи с тем, что операционная система не сразу дает вам возможность получить блок с данными, когда запрошено чтение из файла. При этом ОС поставит запрос в очередь. Из этой очереди задания будут выбираться по различным стратегиям, которые представлены на следующем слайде. После этого происходит ожидание канала. В это время контроллеры должны договориться между собой; после чего контроллер должен принять заявку. И лишь после этого начинается информационная передача.

Если для определения времени доступа объединить все характеристики дисков SSD и HDD, то оно может быть рассчитано по формуле, приведенной на слайде.

Анализируя показанное видно, что, как у дисков HDD, так и у дисков SSD, присутствуют первые два этапа процесса ввода-вывода (ожидание очереди и ожидание канала). Однако, у SSD будут отсутствовать время поиска Seek и время оборота Rotation delay. Вместо нее будет что-то другое, характеризующее доступ к адресной памяти.

Именно поэтому диски SSD значительно «быстрее», и они хорошо себя показывают как на последовательной, так и на случайной нагрузке. Но диски SSD сегодня существенно дороже.



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 длиной 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.

Операционные системы. Часть 3. Память и планирование

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Рассмотрим дисковое планирование в операционных системах.

Рассмотрим случай, когда работающие процессы «нагенерили» некоторое количество запросов на предоставление ввода-вывода, которые необходимо обслужить.

В этом случае операционная система помещает все эти запросы в абстрактную очередь, и начинает их каким-то образом обслужить.

При этом ОС может выбирать различные стратегии для того, чтобы попытаться провести планирование, то есть определить, какие запросы надо обслужить в первую очередь.

Вообще говоря, дисковые планировщики очень похожи на те, которые используются для процессов, но некоторые отличия существуют.

Среди стратегий планирования операций ввода-вывода можно выделить следующие:

- FIFO — «справедливый метод» (означающий, что запросы будут удовлетворяться по мере их прихода), при этом все процессы получат равный доступ к диску. Этот метод выгодно применять, когда поток запросов к диску не очень большой. Если поток запросов к диску у нас начинает расти, то в связи с тем, разные процессы обращаются в разные места диска, то планирование таких запросов по стратегии FIFO, по большому счету, превращается в «случайный планировщик».

- PRI — обслуживание на основе приоритета процесса; при этой стратегии более высокоприоритетный процесс «ставится» в начало очереди. С точки зрения операционной системы этот способ выгоден, т. к. в процессе исполнения приоритеты процесса будут меняться (вспомним хотя бы планировщик FeedBack). Поэтому, и другие процессы тоже получат хороший доступ к диску. Однако, если внимательно посмотреть на статистику, то для коротких заданий эта стратегия ввода-вывода будет работать хорошо, а вот длинные задания (когда надо будет часто общаться с диском) будет работать значительно хуже.

- LIFO (last in first out) — последний запрос, который был поставлен в очередь, будет обслужжен первым. С первого взгляда это достаточно непонятная вещь. Но при применении стратегии LIFO известно, какие из поступивших запросов будут «попадать» рядом друг с другом, и значит, благодаря группировке, в первую очередь будут подвергаться вводу-выводу данные, находящиеся proximity (т. е. близко к друг-другу). И диск при этом будет «меньше двигать головкой», а значит, станет быстрее отдавать данные. Этот принцип удобен для работы с базами данных и в системах обработки транзакций, то есть, когда сообщения короткие и высока вероятность появления «близко-расположенных запросов».

Следует отметить, что все три, описанные выше планировщика, не учитывают информацию о положении «считывающих головок». При этом если в рассмотрение будет приниматься текущее состояние диска и место, где сейчас расположены головки, то можно разработать стратегии планирования, которые будут более эффективно справляться с большим потоком запросов к диску.

Рассмотрим примеры таких стратегий:

- SSTF (Shortest Service Time First) — в этой стратегии анализируется, далеко ли находятся от текущей позиции следующий блок, который необходимо запросить; затем производится сортировка блоков по их близости к нашему положению; после чего головку начинают сдвигать туда, куда ближе;

- SCAN (elevator algorithm) — «алгоритм подъемника» - сводится к тому, что головка в разные стороны движется по диску (к которому есть некоторое количество запросов в очереди) и по удовлетворяет запросы, оказывавшиеся у нее на пути; после того, как головка доходит до конца, она начинает движение в другую сторону. Но легко заметить, что «края» диска при этом обслуживаются гораздо хуже, чем его середина. Поэтому те запросы, которые находятся «на краях», будут обслуживаться медленнее, что несправедливо.

Для ликвидации этой «несправедливости» существует некоторое количество модификаций этого алгоритма.

- C-SCAN — алгоритм заключается в том, что работа с блоками (читаем-записываем) осуществляется только тогда, когда головка движется в одну сторону по диску, а когда она доходит до конца, то она в быстром режиме перепозиционируется в начальное положение. И процесс вновь повторяется;

- N-step-SCAN — разделяет очередь на подочереди длиной N, и обрабатывает одну очередь за один проход; при этом алгоритм еще и проверяет, сколько стоит запросов в каждой из подочередей, и если запросов в очереди мало, то он не будет выполнять эту очередь, а выполнит ее в следующий раз, ожидая, что ОС еще поместит туда заданий;

- FCSCAN — используются две подочереди. И пока одна обрабатывается, другая – заполняется.

Стратегии дискового планирования

В только что рассмотренном примере причина разницы в производительности может быть объяснена продолжительностью поиска. Если выполнение обращений к секторам включает выбор дорожек случайным образом, производительность дискового ввода-вывода окажется чрезвычайно низкой. Для ее повышения нам необходимо уменьшить время, затрачиваемое на поиск дорожки.

Рассмотрим типичную ситуацию в многозадачной среде, когда операционная система поддерживает очередь запросов для каждого устройства ввода-вывода. Соответственно, в очереди одного диска будет находиться некоторое количество запросов на ввод-вывод (чтение или запись) от различных процессов. Если выбирать запросы из очереди случайным образом, то следует ожидать, что искомые дорожки будут располагаться в произвольном порядке, и это приведет к очень низкой производительности. Такое **случайное планирование** может служить точкой отсчета для оценки других методик.

На рис. 11.7 сравнивается производительность различных алгоритмов планирования для примера последовательности запросов ввода-вывода. Вертикальная ось соответствует дорожкам диска; горизонтальная ось соответствует времени, или, что эквивалентно, количеству пройденных дорожек. На этом рисунке предполагается, что изначально головка диска расположена на дорожке 100; диск имеет 200 дорожек, а в очереди запросов к диску содержатся случайные запросы (запрошенные дорожки, в порядке поступления планировщику диска — 55, 58, 39, 18, 90, 160, 150, 38, 184). В табл. 11.2 приведены количественные результаты.

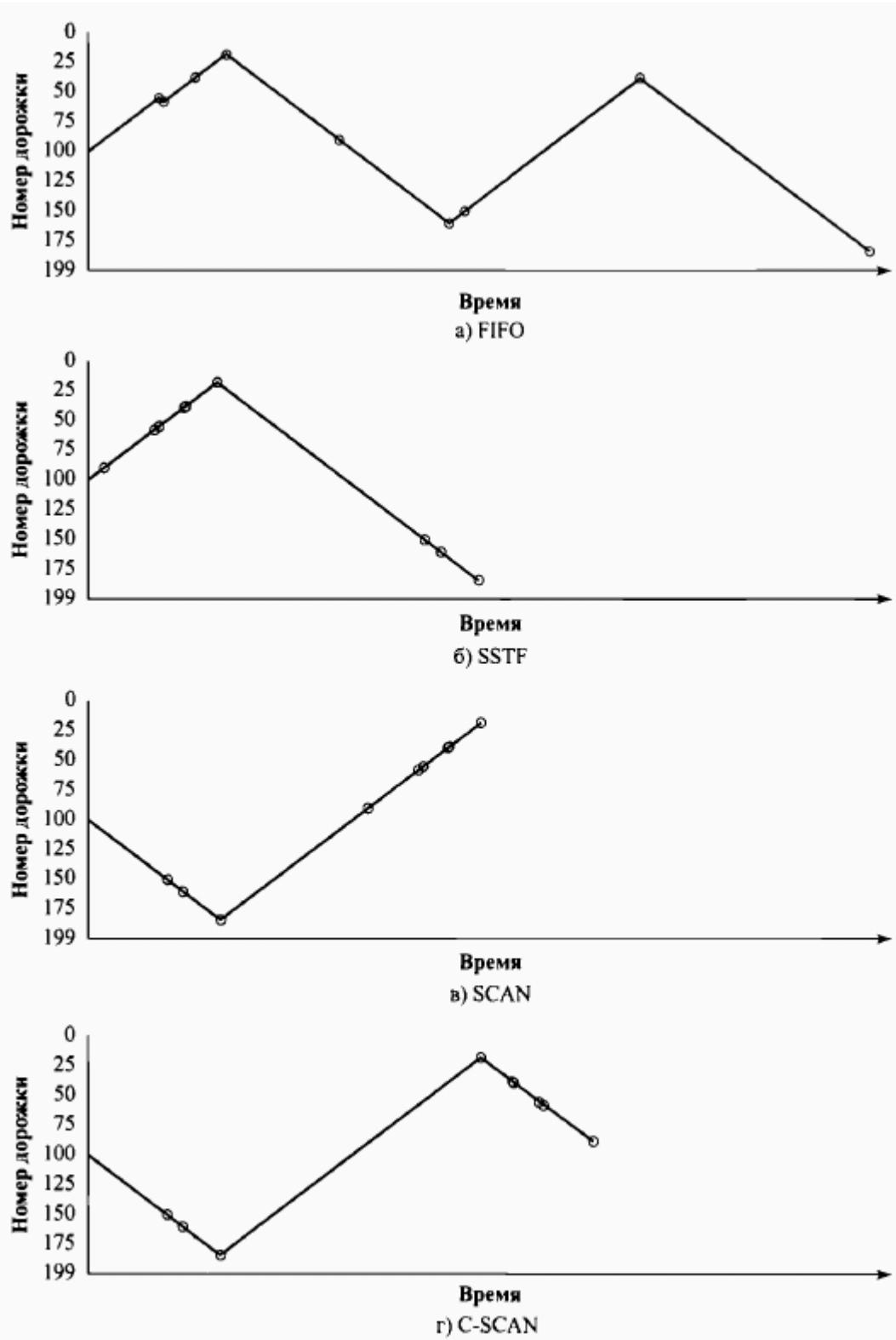


Рис. 11.7. Сравнение алгоритмов дискового планирования (см. табл. 11.2)

Таблица 11.2. Сравнение алгоритмов дискового планирования

а) FIFO (стартовая дорожка 100)		б) SSTF (стартовая дорожка 100)		в) SCAN (стартовая дорожка 100; направление в сторону увели- чения номера дорожки)		г) C-SCAN (стартовая дорожка 100; направление в сторону уменьше- ния номера дорожки)	
Следующая дорожка	Количество пересеченных дорожек	Следующая дорожка	Количество пересечен- ных дорожек	Следующая дорожка	Количество пересечен- ных дорожек	Следующая дорожка	Количество пересеченных дорожек
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Средняя продолжи- тельность поиска	55,3	Средняя продолжи- тельность поиска	27,5	Средняя продолжи- тельность поиска	27,8	Средняя продолжи- тельность поиска	35,8

FIFO

Самая простая форма планирования — планирование “первым пришел — первым вышел” (FIFO), при котором элементы из очереди обрабатываются последовательно. Преимущество этой стратегии — в справедливости, потому что каждый запрос выполняется, причем все запросы выполняются в порядке получения. На рис.11.7, а показано движение головки диска при использовании этой стратегии. Этот график получен непосредственно из данных табл. 11.2, а. Как можно видеть, доступ к диску осуществляется в том же порядке, в котором первоначально были получены запросы.

При использовании стратегии FIFO надеяться на высокую производительность можно только при небольшом количестве процессов и запросах в основном к близким группам секторов. Однако при работе большого количества процессов производительность будет почти такой же, как и при случайном планировании. Таким образом, может быть выгодно рассмотреть более сложную стратегию планирования. Некоторые из таких стратегий перечислены в табл. 11.3 и будут вскоре рассмотрены.

Таблица 11.3. Алгоритмы дискового планирования

Название	Описание	Примечания
Выбор в соответствии с источником запроса		
Random	Случайное планирование	Для анализа и моделирования
FIFO	“Первым вошел — первым вышел”	Наиболее беспристрастный метод
PRI	Приоритет процесса	Очередь запросов к диску управляется извне
LIFO	“Последним вошел — первым вышел”	Максимизация локальности и использования ресурса
Выбор в соответствии с содержимым запроса		
SSTF	Выбор самого короткого времени обслуживания	Высокая степень использования, малые очереди
SCAN	Перемещение вперед и назад по диску	Лучшее распределение обслуживания
C-SCAN	Однонаправленное перемещение с быстрым возвратом	Низкая изменчивость обслуживания
N-step-SCAN	SCAN с N записями в одном пакете	Гарантия обслуживания
FSCAN	N-step-SCAN, где N — размер очереди в начале цикла SCAN	Чувствительный к загрузке

Приоритеты

В системе с использованием приоритета (PRI) управление планированием является внешним по отношению к программному обеспечению управления диском. Такой подход не имеет отношения к оптимизации использования диска, но зато удовлетворяет некоторым другим целям операционной системы. Зачастую коротким пакетным заданиям, а также интерактивным заданиям присваивается более высокий приоритет, чем длинным заданиям, требующим более длительных вычислений. Эта схема позволяет быстро

завершить большое количество коротких заданий в системе и обеспечивает малое время отклика. Однако при использовании этого метода у больших заданий оказывается слишком длительным ожидание выполнения дисковых операций. Кроме того, такая стратегия может привести к противодействию со стороны пользователей, которые будут разделять свои задания на малые подзадания. Не подходит эта стратегия и для работы с базами данных.

Последним вошел — первым вышел

Как это ни удивительно, стратегия выполнения первым последнего запроса имеет свои преимущества. В системах обработки транзакций при предоставлении устройства для последнего пользователя должно выполняться лишь небольшое перемещение указателя последовательного файла. Использование преимуществ локальности позволяет повысить пропускную способность и уменьшить длину очереди. К сожалению, если нагрузка на диск велика, существует очевидная возможность голодания процесса. Когда задание размещает запрос ввода-вывода в очереди, и оно попадает в ее начало, это задание не сможет продолжать работу, пока вся очередь перед ним не опустеет.

Три рассмотренные стратегии планирования — FIFO, PRI и LIFO — основаны исключительно на атрибутах очереди или запрашивающего процесса. Однако если планировщику известна текущая дорожка, то появляется возможность использования стратегии планирования, основанной на содержимом запроса.

SSTF

Стратегия выбора наименьшего времени обслуживания (Shortest Service Time First — SSTF) заключается в выборе того дискового запроса на ввод-вывод, который требует наименьшего перемещения головок из текущей позиции. Следовательно, мы минимизируем время поиска. Естественно, постоянный выбор минимального времени поиска не дает гарантии, что среднее время поиска при всех перемещениях будет минимальным; тем не менее эта стратегия обеспечивает лучшую по сравнению с FIFO производительность дисковой системы. Поскольку головки могут перемещаться в двух направлениях, при равных расстояниях для принятия решения может быть использован случайный выбор направления.

На рис. 11.7, б и в табл. 11.2, б показана производительность стратегии SSTF для той же последовательности запросов, что и при рассмотрении стратегии FIFO. Первое обращение выполняется к дорожке 90, потому что это самая близкая к начальной позиции запрошенная дорожка. Следующей дорожкой, к которой будет доступ, является 58, потому что она самая близкая из оставшихся запрошенных дорожек к текущей позиции 90. Последующие дорожки выбираются аналогично.

SCAN

Все стратегии, описанные к настоящему времени (за исключением FIFO), могут оставить некоторый запрос невыполненным до тех пор, пока не освободится вся очередь, т.е. в ходе работы всегда могут иметься новые запросы, которые будут выбраны до уже имеющегося в очереди. Избежать такого рода голодания можно при использовании стратегии SCAN, известной также как алгоритм лифта (Elevator) из-за работы, напоминающей работу лифта.

При использовании этого алгоритма перемещение головки происходит только в одном направлении, удовлетворяя те запросы, которые соответствуют выбранному направ-

лению. После достижения последней дорожки в выбранном направлении (или когда исчерпаются возможные запросы), направление изменится на противоположное. Это последнее уточнение иногда называют стратегией LOOK. Затем направление обслуживания меняется на противоположное, и сканирование продолжается в противоположном направлении, снова собирая все запросы по порядку.

Стратегия SCAN представлена на рис. 11.7, в и в табл. 11.2, в. В предположении, что начальным направлением является увеличение номера дорожки, первая выбранная дорожка равна 150, поскольку эта дорожка — самая близкая к начальной дорожке номер 100 в направлении увеличения.

Как видите, стратегия SCAN ведет себя почти так же, как и стратегия SSTF. Фактически, если предположить, что изначально головка перемещается в сторону меньших номеров дорожек, схема планирования окажется идентичной для SSTF и SCAN. Однако это статический пример, в котором в очередь не добавляется ни один запрос. Однако даже при динамическом изменении очереди стратегии SCAN и SSTF выглядят, как правило, очень похоже.

Обратите внимание, что стратегия SCAN имеет отрицательный “перекос” в отношении недавно пройденной области. Таким образом, данная стратегия использует имеющуюся локальность не так эффективно, как SSTF.

Нетрудно увидеть, что стратегия SCAN оказывает предпочтение тем заданиям, запросы которых относятся к дорожкам, находящимся ближе всего к центру либо наиболее удаленным от него, а также отдает предпочтение запросам, поступившим последними. Первой проблемы можно избежать путем применения стратегии C-SCAN; вторая же проблема решается с помощью стратегии N-step-SCAN.

C-SCAN

Стратегия C-SCAN (циклическое сканирование) ограничивает сканирование только одним направлением. Когда обнаруживается последняя дорожка в заданном направлении, головка возвращается в противоположный конец диска, и сканирование начинается снова. Это уменьшает максимальную задержку, вызванную новыми запросами. Если при использовании стратегии SCAN ожидаемое время сканирования от внутренней к внешней дорожке равно t , то ожидаемый интервал обслуживания секторов, находящихся на периферии, будет равен $2t$. При использовании стратегии C-SCAN этот интервал будет порядка $t + s_{\max}$, где s_{\max} — максимальное время поиска.

Поведение стратегии C-SCAN показано на рис. 11.7, г и в табл. 11.2, г.

В этом случае первые три запрошенные дорожки — 150, 160 и 184. Затем начинается сканирование с самого малого номера дорожки, так что следующая запрошенная дорожка — 18.

N-step-SCAN и FSCAN

При использовании стратегий SSTF, SCAN и C-SCAN может возникнуть ситуация, когда один или несколько процессов с высокой частотой обращений к одной дорожке монополизируют устройство за счет многочисленных повторений запросов к одной и той же дорожке. Наиболее характерна эта особенность для многоповерхностных дисков с большой плотностью записи. Для предотвращения такого “залипания головки” очередь дисковых запросов может быть сегментирована, причем за один прием полностью выполняется весь сегмент заданий. Примерами такого подхода являются стратегии N-step-SCAN и FSCAN.

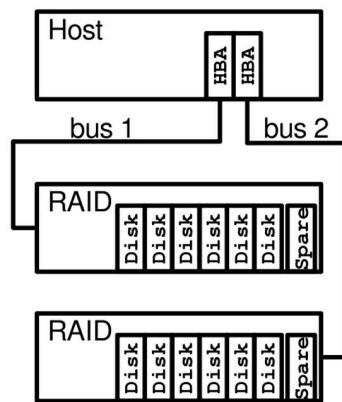
Стратегия *N*-step-SCAN разделяет очередь дисковых запросов на подочереди длиной *N*. Каждая подочередь обрабатывается за один прием с использованием стратегии SCAN. В ходе обработки очереди к некоторой другой очереди могут добавляться новые запросы. Если в конце текущего сканирования доступными оказываются менее *N* запросов, то все они обрабатываются в следующем цикле сканирования. При больших значениях *N* производительность алгоритма *N*-step-SCAN достигает таковой у алгоритма SCAN; предельный случай *N* = 1 соответствует стратегии FIFO.

FSCAN — стратегия, использующая две подочереди. С началом сканирования все запросы находятся в одной из очередей; другая при этом остается пустой. Во время сканирования первой очереди все новые запросы попадают во вторую очередь. Таким образом, обслуживание новых очередей откладывается, пока не будут обработаны все старые запросы.

60. Концепции RAID



• RAID (Redundant Array of Independent Disks)



- Ввод вывод к нескольким дискам может происходить параллельно
- С увеличением количества дисков надежность падает
- RAID — несколько дисков, которые ОС логически объединяет в один
 - Данные распределены по всем дискам
 - Может использоваться дополнительная (избыточная) емкость для хранения контрольной информации
 - Один блок данных может находиться на нескольких дисках
 - Один запрос может также быть выполнен параллельно
 - Необходимо учитывать производительность адаптеров и шин подключения к адаптерам, а также шины I/O (PCI, PCIe)

Потребность в постоянном увеличении производительности и дисковой памяти; производительность вторичных хранилищ (диски) растёт медленнее первичных (CPU, ОЗУ).

JBOD (Just a Bunch of Disks) - массив независимых и параллельно работающих жестких дисков, по которым единое логическое пространство распределено последовательно. По SCSI через один HBA (Host Bus Adapter) можно подключить 15 дисков.

Много процессоров -> Обращение к разным дискам -> Повышение скорости IO

Больше дисков -> Меньше надёжность (Выше вероятность отказа диска)

RAID (Redundant Array of Independent Disks) - набор дисков, объед. в логический том.

- 1) Данные распределены по всем дискам.

- 2) Могут быть нужны доп. диски для контрольной инфы (восст. данных при отказе).
- 3) Дублирование данных (один диск умер, второй жив).
- 4) Запрос может выполняться параллельно (если блок раскинут по нескольким дискам).
- 5) Нужны мощные адаптеры и шины передачи данных.

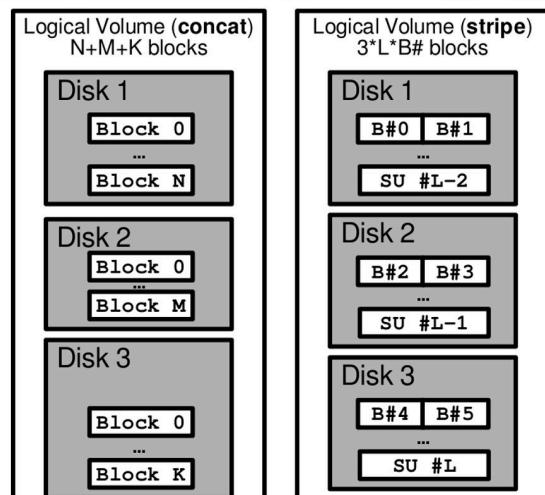
Один блок на нескольких дисках, параллельность -> Увеличение производительности
Скорость шины < diskCount * diskDTR -> Бутылочное горлышко



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

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

RAID 0 — concatenation/striping



- Объединяет диски в один том большего размера
- Конкатенация — можно склеивать диски разного размера
- Страйп — одинакового размера или размера меньшего диска
 - Stripe Unit — объединение блоков общим размером Stripe Size (16-128Кб)
 - SU зависит от конфигурации при создании
- Характеристики
 - Низкая надежность
 - Высокая пропускная способность
 - Высокая скорость обработки запросов для чтения и записи (stripe)

Операционные системы. Часть 3. Память и планирование

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

Используется, когда производительность, емкость и малая стоимость важнее надежности.

Больше дисков -> Больше отказов

Concatenation - диски разного размера объед. в один том, послед. нумерация от 0 до N+M+K -> При отказе диска данные на остальных будут живы, производительность не растет (В Столлингсе отсутствует)

Striping - данные делятся на Stripe Units (на слайде - 2 блока) и раскидываются по дискам -> При обращении к нескольким блокам велика вероятность, что они на разных дисках -> Повышение скорости IO, но при отказе диска большими данным хана.

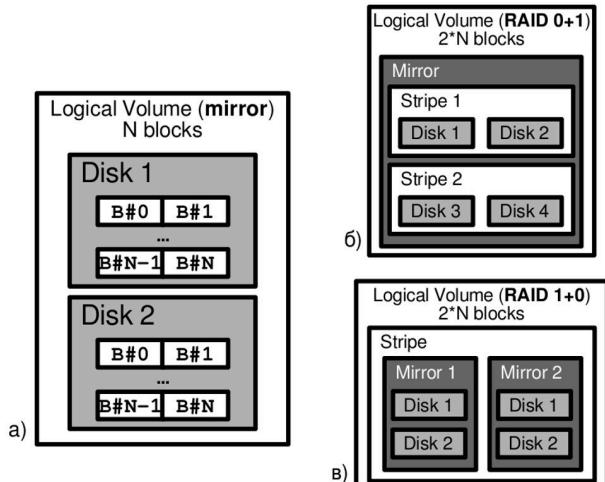
Характеристики:

- Низкая надежность (отказ диска -> смерть многих данных)
- Высокая пропускная способность передачи данных
- Высокая скорость обработки запросов IO (для Stripe)

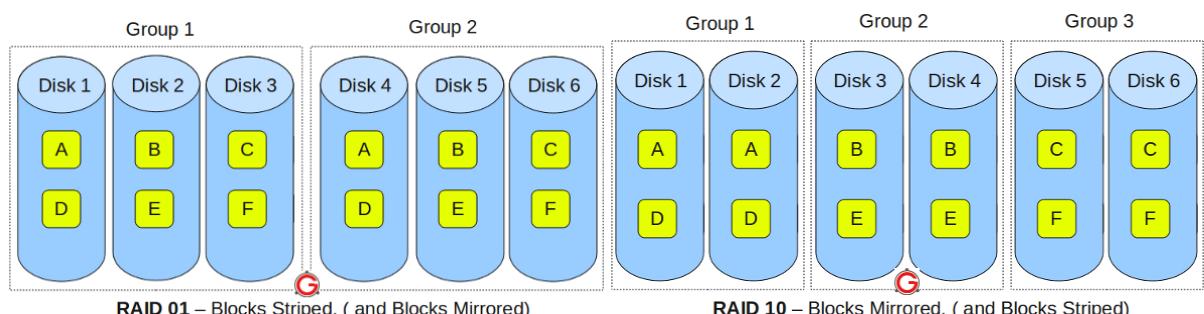
Категория	Уровень	Описание	Требуемое количество дисков ¹	Доступность данных	Пропускная способность передачи больших данных	Скорость запросов малого ввода-вывода
Отражение ²	1	Отражение	$2N$	Больше, чем у RAID 2, 3, 4 и 5, но меньше, чем у RAID 6	Для чтения больше, чем у одного диска; для записи сравнивается с одним диском	Для чтения почти вдвое больше, чем у одного диска; для записи сравнивается с одним диском

Raid 1 - Зеркалирование

- Объединяет диски, копируя блоки
- Разные конфигурации:
 - RAID 0+1 (б) RAID 10 (в)
- Двойная (или больше) избыточность
- Характеристики (а):
 - Высокая надежность
 - двойная пропускная способность на чтение и одинарная на запись
 - Двойная скорость обработки запросов для чтения и одинарная на запись
- Характеристики у (б) и (в) комбинируются с RAID 0.



Решение проблемы надежности RAID 0 путем дублирования данных (зеркалирования). (В Столлингсе RAID 0+1)



RAID 0+1 - RAID 1 из RAID 0 (страйпим данные по дискам, потом зеркалируем)

Отказ диска -> т.к. группы большие, страйп валится целиком; надо его вырубать, заменять диск, собирать новый страйп и синхронизировать с зеркалом

RAID 10 - RAID 0 из RAID 1 (зеркалируем, потом страйпим)

Отказ диска -> т.к. группы маленькие, валится только диск; заменяем его и синхронизируем

В качестве плюсов можно выделить:

- простоту восстановления данных,
- высокую скорость и пропускную способность при чтении (выбирается диск, у которого минимальная задержка из-за вращения и минимальное время поиска).
- параллельное обновление записи на обоих дисках (таким образом время определяется скоростью более медленной операции)
- надежность (уступает только RAID 6; у 10 выше, чем 0+1)

Минус - количество дисков возрастает в 2 или более раза, поэтому RAID 1 используется только для самых важных файлов.

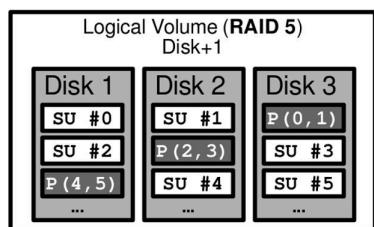
RAID 1 лучше подходит для чтения и может превышать скорость RAID-0 в два раза в среде, ориентированной на транзакции.

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

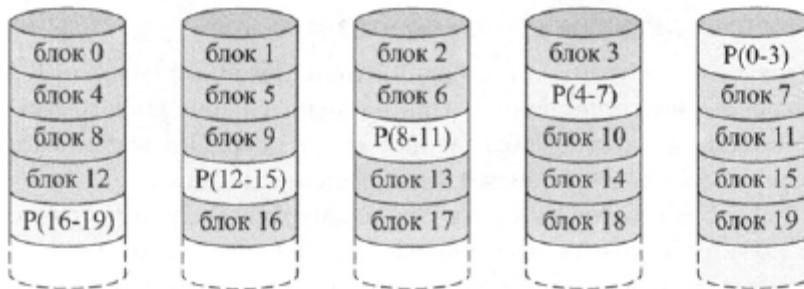
Категория	Уровень	Описание	Требуемое количество дисков ¹	Доступность данных	Пропускная способность передачи больших данных	Скорость запросов малого ввода-вывода
Независимый доступ	4	Четность с чередующимися блоками	N+1	Гораздо выше, чем у одного диска, сравнимо с RAID 2, 3 и 5	Для чтения аналогична RAID 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



RAID 4, RAID 5, RAID 6



- Подсчитывает четность для SU
 - $P(0,1) = SU(0) \oplus SU(1)$
 - Если сбой диска 2 $SU(1) = P(0,1) \oplus SU(0)$
- 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



с) RAID 5 (распределенная четность с чередующимися блоками)

RAID 5 - диски страйпятся так, чтобы можно было считать корректирующие коды четности, после чего исходные данные и четность размазываются по разным дискам. Слайд: три диска, два страйпа и их XOR. Последний с каждым уровнем смещается на другой диск.

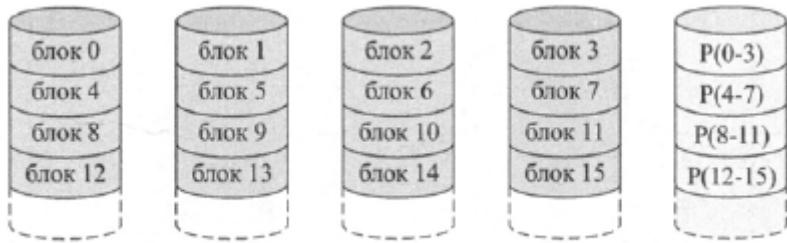
При сбое одного диска можно будет вычислить четность XOR'ом страйпов других дисков, либо страйп XOR'ом четности и другого страйпа. Система даже может так работать, но очень медленно.

Высокая надежность

Для распределения четности требуется дополнительный диск

В нормальном состоянии быстрое чтение данных, при сбое резко падает

Медленная скорость записи (вычисление четности)



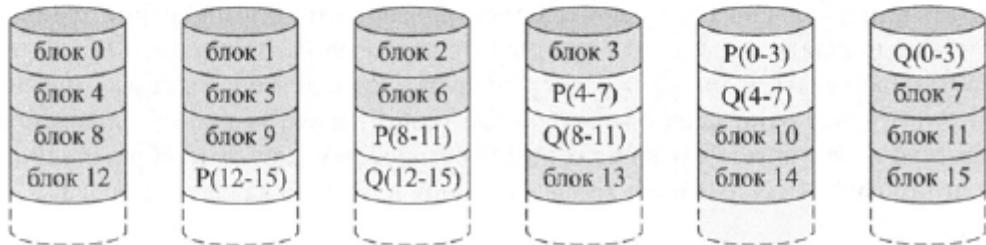
д) RAID 4 (четность с чередующимися блоками)

RAID 4 - четность пишется на отдельный диск, а не размазывается.

Так как диск один, запись еще медленнее, чем у RAID 5

Надежность такая же высокая

Тоже один дополнительный диск



ж) RAID 6 (двойная распределенная четность с чередующимися блоками)

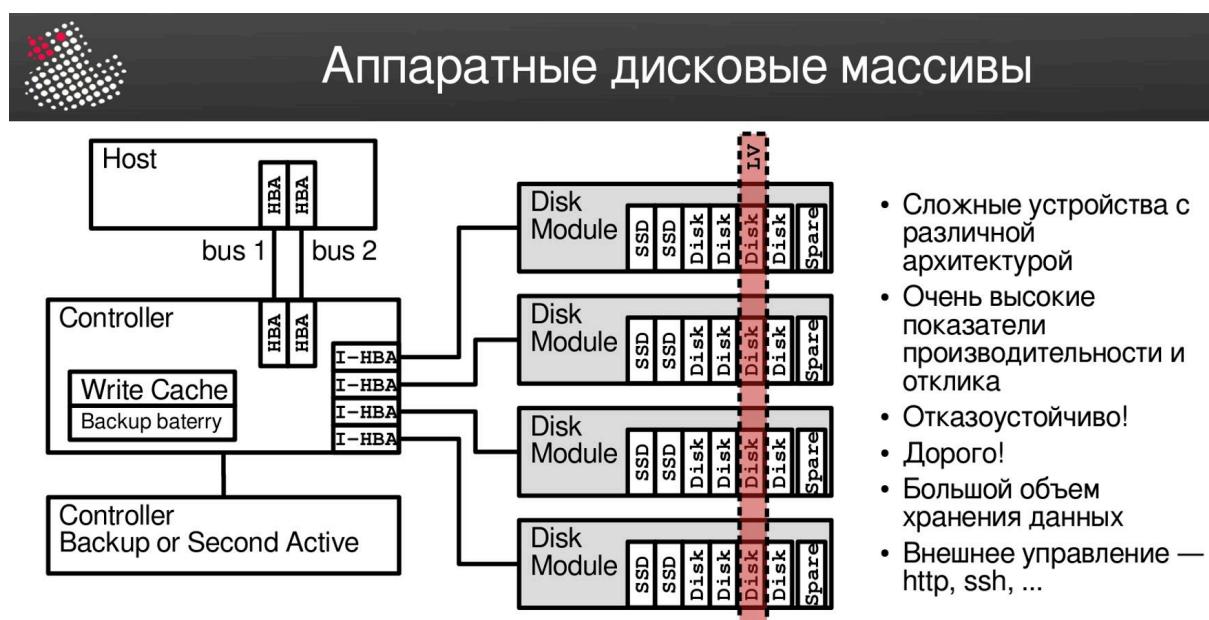
RAID 6 - двойная четность (XOR и еще дополнительный алгос)

Два дополнительных диска, четность распределена (как в RAID 5)

Чрезвычайно высокая надежность хранения данных (потери возможны только при одновременном отказе трех дисков)

Очень медленная запись (на треть хуже RAID 5)

Быстрое чтение



Сложная конструкция RAID-массивов, используется в облачных хранилищах. Состоит из контроллеров со спец. микросхемами для расчета четности (обл. быстрым IO) и внутренних адаптеров HBA (Host Bus Adapter), к которым подключаются накопители (Disk Module) с дисковыми полками.

Контроллеры обычно дублируются для надёжности (внизу вторичный контроллер, тоже подкл. к дисковым полкам). У них есть буфер записи - память для быстрого сохранения записываемых польз. программой транзакций, которые потом разносятся контроллерами по томам. Но при сбоях питания она умрет, потому что строится на DIMM (вид DRAM), поэтому в контроллерах ставят резервные батареи.

Скорость передачи данных нужно архитектурно рассчитывать. Иногда контроллеры подключают к хосту большим числом HBA, чем возможно, чтобы занять все полосы пропускания, потому что из-за количества дисков HBA и шины могут не справляться и становиться единой точкой отказа/бутылочным горлышком.

Для каждого модуля организуется отдельное питание, и он может вылететь целиком со всеми дисками, поэтому логические тома создаются "вертикально", чтобы выход модуля из строя не убил всю ФС. Если нужен больший объем, можно пристыковать следующую "полосу".

Такие массивы дорогущие (не меньше 500к), но отказоустойчивые, быстрые, вместительные и расширяемые.

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



Файл

- 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

Долгосрочное существование - файл хранится на диске и не исчезает при выходе пользователя из системы.

Совместное использование процессами - файлы имеют имена и права, обеспечивающие управляемый совместный доступ.



Поле, запись, файл, база данных

- 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)

Формально поле может быть набором значений (string), характеризуется длиной и типом данных.

Запись - набор связанных между собой полей. Си - структура, БД - кортежи. Длина поля и/или их кол-во м.б. переменными.

Файл - именованный набор однородных (логически взаимосвязанных) записей (или полей).

БД - набор файлов с явно выраженнымми отношениями полей и записей.

Пример влияния операций на структуру - индексация для быстрого поиска.

Базовая файловая структура в Unix(-like) представляет собой поток байтов. Текст программы на Си хранится в виде файла, но не имеет физ. полей и записей.



Подсистема управления файлами. Задачи ОС

- Возможность хранить файлы и выполнять пользователю над ними операции:
 - 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

Операционные системы. Часть 3. Память и планирование

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

Подсистема управления файлами - часть ОС, позволяющая хранить файлы и проводить над ними операции:

Управление доступом - создатель назначается владельцем и может задавать права на использование файла.

Резервное копирование - обычно не явл. частью ФС, а ложится на отдельные приложения, где юзер может выбрать место и время бэкапа.

Работа с файлом по имени - возможность выбрать символьное имя (с цифровыми тяжело).

Гарантия корректности данных - содержимое файла не должно меняться во времени (поэтому для важных файлов делают несколько контр. копий и считают контр. сумму, например, с md5).

Приемлемая производительность - вторичная память очень медленная, поэтому для работы с файлами применяется буферизация, RAID и т.д.

Различные типы устройств хранения - флешки, дискеты и т.д.; требуются драйвера, способные писать блоки из буфера на диск или другое устройство с учетом геометрии устройств и конвертации адресов блоков.

Минимизация потерь и повреждения данных - обеспечение целостности данных при сбоях процессоров и ОС.

Базовый набор функций IO - ядро должно торчать наружу сисколами для IO, на которых работают более высокогоуровневые инструменты.

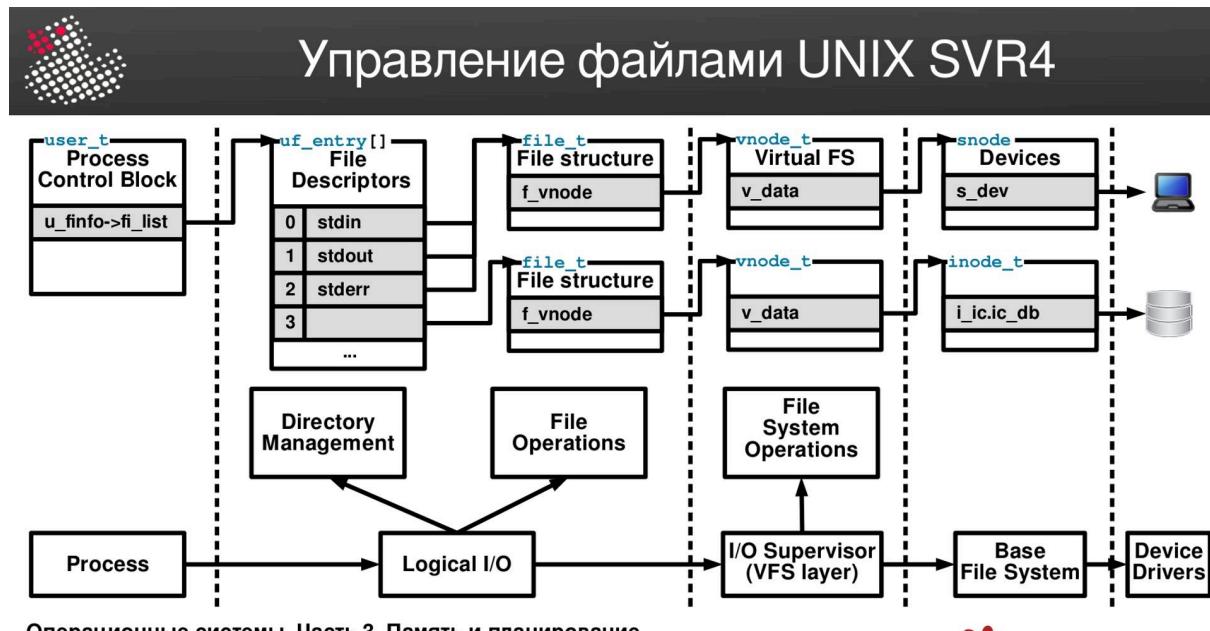
Совместное использование файлов - чтобы файл не менялся потоками/процессами одновременно, используют блокировки разного уровня (файла, записи).



Совместное использование файлов

- Владелец — пользователь, имеющий все права на файл, и распределяющий эти права:
 - Отсутствие прав — другие пользователи не знают о существовании файла
 - Знание — пользователи знают о существовании файла, и могут запросить права у владельца
 - Выполнение — пользователь может загрузить и выполнить программу
 - Чтение, добавление в конец файла, модификация, удаление
 - Изменение прав доступа
- Доступ к файлу может быть передан разным наборам пользователей:
 - Конкретному пользователю
 - Списку пользователей или группе пользователей
 - Всем пользователям
- Блокировка одновременного доступа
 - Всего файла
 - Отдельных записей

64. Управление файлами в UNIX SVR4



Операционные системы. Часть 3. Память и планирование
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

Некоторый процесс генерирует вызовы ядра.

В ядре есть подсистема логического IO, у которой есть части, отвечающие за управление каталогами и файловые операции. Она отвечает за переход в каталог, открытие файла по имени, чтение, запись и т.д.

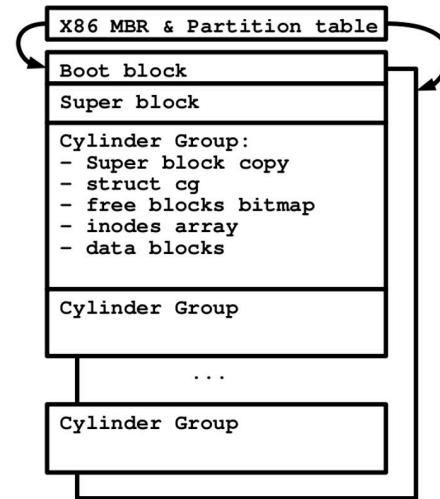
Уровень супервизора ввода-вывода стандартизирует интерфейс взаимодействия (виртуальную ФС) внутри всех ФС (объектно-ориентированный подход). В нём существуют виртуальные узлы, относящиеся к конкретным используемым ФС.

Base File System зависит от драйверов устройств и физического расположения блоков на диске.

В структуре, описывающей PCB, есть user_t - часть, где находится информация о пользователе, содержащая u_info->fi_list (uf_entry[]) - список значений дескрипторов открытых процессом файлов и стандартных потоков ввода-вывода (file_t). Они указывают на виртуальные узлы, под которыми идут конкретные реализации для конкретных ФС (snodes - специальные устройства. inode - записи файлов).

Структура файловой системы **UNIX SVR4 UFS**

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



На x86 в начале диска лежат MBR и таблица разделов дисков, внутри которых может быть некоторое кол-во томов логических записей.

Логический том - последовательность блоков, выделенная на диске для ФС.

(На всякий случай) inode - индексный узел/дескриптор, содержащий всю метаинформацию о файле, кроме имени (кол-во ссылок, размер, даты, тип).

Каждый раздел содержит:

- Boot block - загрузчик ОС
- Суперблок - блок с геометрией раздела, кол-вом разделов, бэкапами, служ. параметрами и т.д.
- Цилиндровые группы - последовательно расположенные сектора равного размера, содержащие копии суперблока (для восстановления ФС при сбоях диска), описание ЦГ, информацию о свободных блоках, массив inode и блоки данных. Таким образом, файлы и их данные хранятся локально, что позволяет головке диска меньше двигаться и делает ФС выгодной с точки зрения производительности.

+ смотреть [69 билет](#)

65. Каталоги файлов. Элементы каталога, операции ОС.



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

Операционные системы. Часть 3. Память и планирование

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Каталоги позволяют организовать наборы файлов и управление ими, храня соответствия имён файлов и inode. При этом в Unix, в отличие от Windows, дополнительная информация хранится в записях файла на диске.

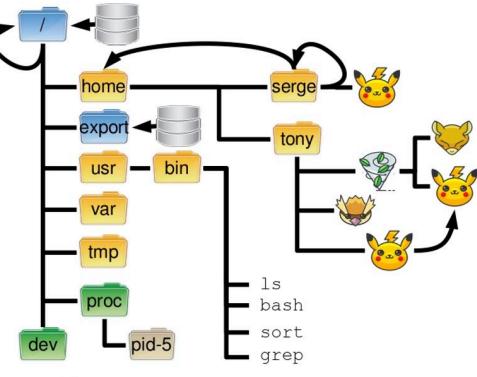
Классификация информации, представленной в каталоге:

- Основные элементы:
 - Имя;
 - Тип (бинарный, текстовый и т.д.);
 - Организация внутренней структуры; опред. по magic number (набор байт в начале файла) или части имени после точки (Windows - расширение; в Unix юзается, но отсутствует как понятие)
- Адресная информация:
 - Том - указатель на физ. устройство;
 - Нач. адрес - номер первого блока;
 - Занятый и зарезерв. размер - текущ. и макс. кол-во байт.
- Информация об управлении доступом:
 - Владелец (юзер, управл. файлом)
 - Разрешённые действия (rwx)
- Информации об использовании:
 - Дата создания и создатель (право могло уйти другому юзеру)
 - Дата последнего чтения и последний читатель
 - Дата последнего изменения и последний редактор
 - Дата последнего резервного хранения на другом устройстве
 - Текущее использование (открыт ли сейчас, кто читает/пишет и т.д.)



Каталог, операции ОС

- Часть элементов каталога может находиться в записи о файле, уменьшая его размер
- Каталог может быть полностью или частично загружен в основную память
- Простейшая структура — список записей фиксированной длины
- 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

Операционные системы. Часть 3. Память и планирование

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

Часть элементов каталога может находиться в заголовочной записи файла (дата, владелец и др. - поля struct inode) -> меньше информации в каталоге -> легче кэшировать в памяти -> выше скорость.

Большое количество файлов внутри одного каталога сильно снижает производительность.

Основные операции на слайде.

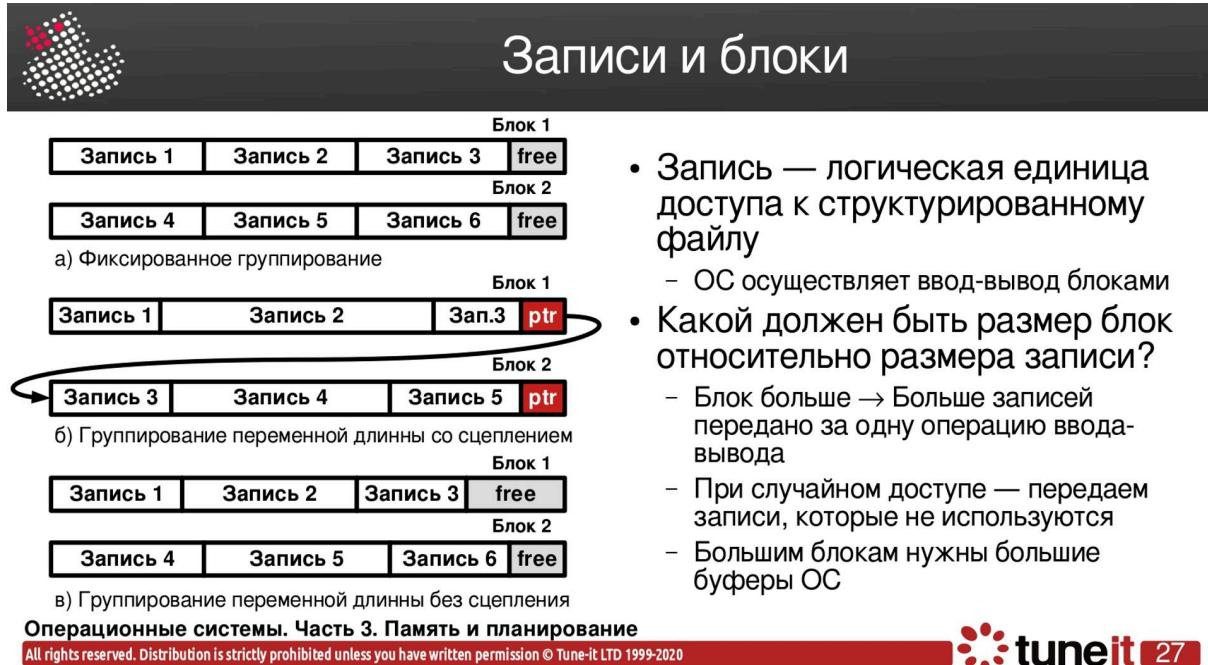
Простейшая структура каталога - файл-словарь со списком записей фикс. длины и именами файлов в качестве ключей. Это юзалось в ранних ОС, но было неудобно.

Потом стали использовать подкаталоги -> Многоуровневость -> Несколько деревьев (Windows) / Связный граф (Unix, за счёт симлинков)

В разных каталогах можно иметь одинаковые имена файлов (но абс. пути будут уникальны).

В Unix и Windows всегда есть корень ФС и традиционная архитектура каталогов, но в Unix есть диски подключаются к точкам монтирования, а в Windows “назначаются” на отдельные буквы.

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

Запись - логическая единица доступа к структурированному файлу. ОС осуществляет ИО блоками, в которых сгруппированы записи.

Задача: сгруппировать записи в файле так, чтобы ОС работала с блоками эффективно

- 1) Фиксированное группирование.

Забивание блока записями одинаковой длины. Если запись не помещается, то переходит в следующий блок, а в предыдущем остается свободное место. Тогда для чтения одной записи достаточно прочитать один блок.

- 2) Группирование переменной длины со сцеплением.

Забивание блока записями переменной длины. Если запись не помещается, то часть переходит в следующий блок, часть (с указателем) остается в предыдущем. Так как читать надо два блока и доступ может быть случайным (непоследовательным), производительность может падать.

- 3) Группирование переменной длины без сцепления.

Аналогично пункту 1, но длина записей является переменной. Более чревато внутренней фрагментацией данных.

Какой размер блока должен быть относительно размера записи? Чем больше блок...

- 1) Тем больше записей передается за одну операцию ИО. Чем меньше операций и больше передаваемых данных, тем лучше.
- 2) При случайном доступе тем больше "лишних" записей придётся читать.
- 3) Тем больше должны быть буфера ФС для промеж. хранения данных.

Поэтому повышение размера блока невозможно бесконечно. Поэтому для оптимизации операций ИО:

- 1) Для ОС выбирают среднее, оптимальное значение блока, обычно кратное размеру страницы.
- 2) Программист может подобрать оптимальные размеры записей под блок ОС.

Размещение файлов

Номер блока на дорожке							
Дорожка	0	1	2	3	4	5	6
0							
1							
2							
3							
4							
5							
6							
7							
N							

- 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 влияют на характеристики:
 - Внутреннюю и внешнюю фрагментацию
 - Частота выделения (allocation) блоков для всей файловой системы
 - Время выделения блоков
 - Размер служебной информации о размещении файлов

Диск реализован в виде блоков, расположенных на дорожке.

Как наиболее просто организовать ФС из блоков на диске?

- 1) Надо ли сразу выделять пространство для файла? (предварительное или динамическое размещение)
Можно быстро выделить всё пространство путем одной записи, но файлу будет тяжело расти.
- 2) Какой размер порции выбрать для чтения и записи? (Порция - непрерывная единица размером от блока до файла)
- 3) Должен быть этот размер фиксированным или переменным?
- 4) Какой тип структур данных или таблиц используется для учета порций файла?
(например, FAT)

Решения (ответы на эти вопросы) влияют на следующие характеристики:

- 1) Внутреннюю и внешнюю фрагментацию.
- 2) Частоту выделения блоков для ФС.
- 3) Время выделения блоков.
- 4) Размер служебной информации о размещении файлов.
- 5) Объем информации, которую ФС может хранить.
- 6) Необходимость компрессии данных при записи.

Из-за большого количества факторов, которые надо учитывать (принимаемых решений и результирующих характеристик) проектирование качественной и производительной ФС - очень сложная задача.

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



- Особенности

- Предварительное выделение пространства для всего файла.
- Один уровень каталога
- Размер блока 512 байт

- Характеристики:

- Внутренняя фрагментация мала
- Внешняя может быть большой — необходимо обязательное сжатие ФС
- Выделение блоков для файла производится однократно
- Невысокое время выделения блоков — если ФС пуста, то минимальное; если заполнена, то возможны ошибки размещения
- Минимальный размер служебной информации о размещении файлов

	Непрерывный
Предварительное размещение	Необходимо
Фиксированный или переменный размер порции	Переменный
Размер порции	Большой
Частота размещения	Одинарное размещение
Время размещения	Среднее
Размер таблицы размещения файла	Одна запись

Особенности:

- Предварительное выделение непрерывного пространства для файла, в момент его создания требуется объявлять размер
- Существует только один уровень каталога (без подкаталогов)
- Размер блока - 512 байт

Характеристики:

- Малая внутренняя фрагментация (в блоке не очень много свободного места)
- Большая внешняя фрагментация (дыры на месте удаляемых файлов -> необходимо сдвигать занятые блоки в начало)
- Однократное выделение блоков для файла
- Высокая скорость чтения и записи (т.к. блоки располагаются последовательно)
- Невысокое время выделения блоков (очень быстро, если ФС пуста; медленнее и чревато ошибками, если забита)

- Минимальный объем служебной информации (т.е. почти всё место под пользовательские данные)

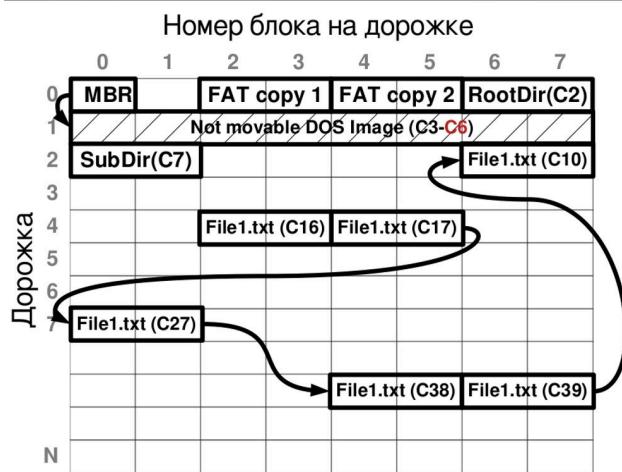
Порядок: MBR (Master Boot Record, запись для загрузки ОС), каталог из нескольких блоков и файлы.

Каждому файлу в таблице размещения соотв. только один элемент, определяющий нач. блок и длину файла.

68. Цепочечное размещение файлов (на примере DOS FAT)

	Цепочечный
Предварительное размещение	Возможно
Фиксированный или переменный размер порции	Фиксированные блоки
Размер порции	Малый
Частота размещения	От низкой до высокой
Время размещения	Длительное
Размер таблицы размещения файла	Одна запись

Цепочечное размещение файлов (DOS FAT)



- Особенности

- Выделение пространства для файла по мере необходимости
- Размер порции файла — кластер (C) — от 512 (FAT12) до 32Кб (FAT32)
- Запись каталога содержит имя (8+3), размер файла, ..., номер начального кластера: File1.txt
- FAT (File Allocation Table) — Содержит таблицу аллокации цепочки кластеров

CL#	10	16	17	27	38	39	39
Next	FFFF	17	27	38	39	10	
Cl#							

- Характеристики:

- Внутренняя фрагментация мала
- Внешняя может быть большой — файлы размещены непоследовательно, необходимо обязательное дефрагментирование ФС для увеличения скорости
- Высокое время выделения блоков файла
- Размер служебной информации о размещении файлов ограничен FAT

Появилась в DOS, используется до сих пор на флешках. FAT = File Allocation Table (т.е. не ФС, а часть).

Иллюстрация на слайде: справа таблица кластеров, 16 -> 17 -> 27 -> 38 -> 39 -> 10 -> NULL (конец цепочки).

В начале - MBR, 2 копии FAT (т.к. в единичном экземпляре при смерти FAT сдохнет сразу всё - информации о размещении не будет), корневая директория, другие директории и файлы.

С целью размещения большего числа информации стали увеличивать размер кластера -> При хранении большого количества мелких (меньше 32 Кбайт) файлов будет большая внутренняя фрагментация.

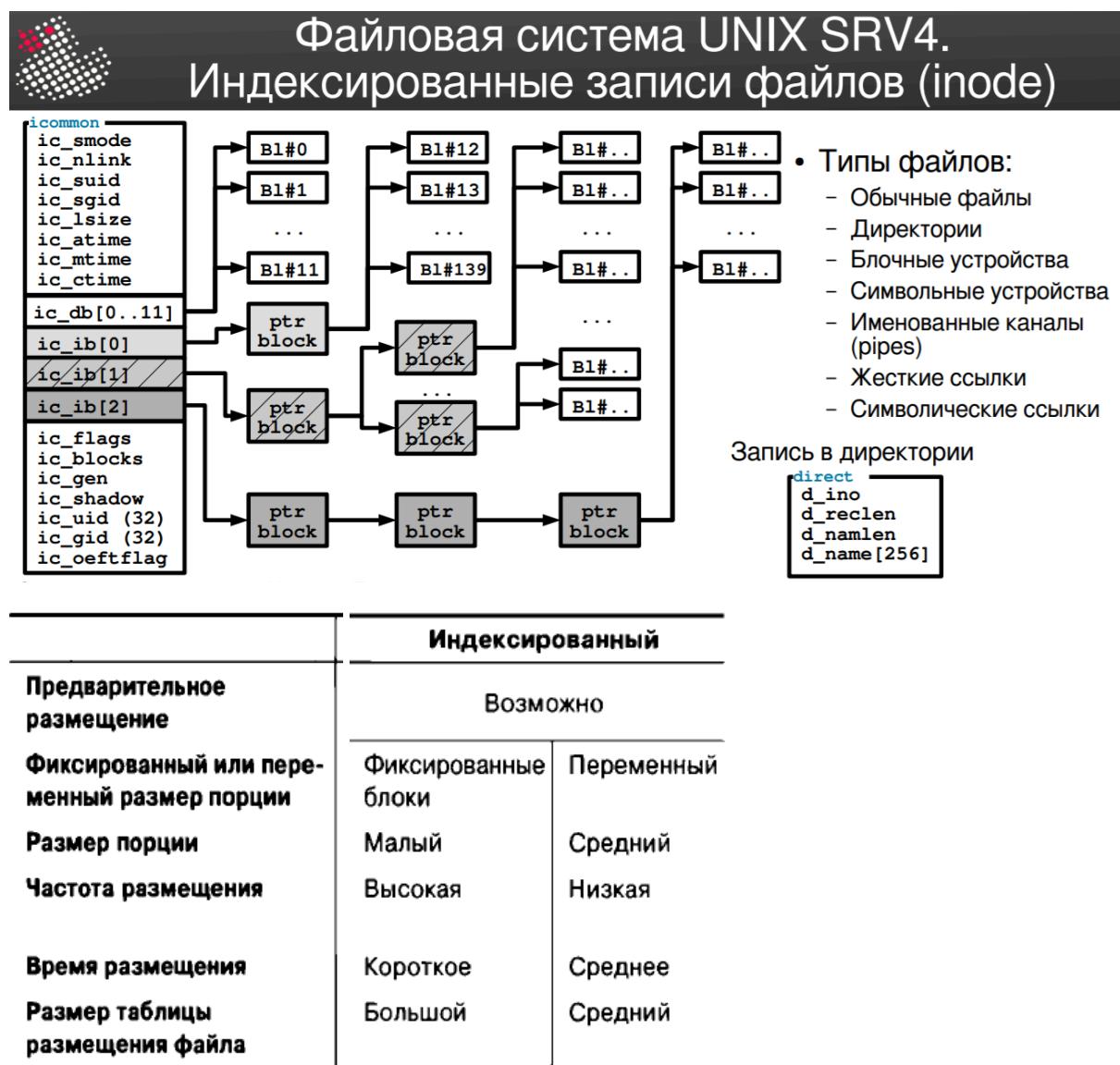
Особенности:

- Пространство под файл выделяется не однократно, а по мере необходимости
- Размер блока - от 512 байт (FAT 12) до 32 Кбайт (FAT 32); 12, 16, 32 - количество бит внутри записи FAT под номер кластера
- Запись каталога содержит имя (8 на имя + 3 на расширение), размер файла, ..., номер начального кластера: File1.txt

Характеристики:

- Малая внутренняя фрагментация
- Файлы перемешиваются между собой -> Низкая производительность, нужно дефрагментировать (размещать последовательно)
- Внешняя фрагментация отсутствует (можно раскидать блоки)
- Высокое время выделения блоков файла
- Размер пространства ограничен FAT (сколько может быть записей в таблице, столько может быть кластеров)

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



На слайде слева - как представлен один **файл** в нашей файловой системе. По факту - это набор информационных полей (размер, количество ссылок, время создания, флаги и т.п. (то есть по факту что выводится при `ls -l`)) и 2 массива. В массивах этих хранится как раз содержимое нашего файла, но каким же образом? Первый массив, `ic_db` (`direct block`) хранит непосредственно 12 блоков, в которых расположено “начало файла”, сколько поместится. Если не поместилось, идем во второй массив, `ic_ib` (`indirect block`), в котором есть всего 3 элемента, соответствующие 3-м уровням косвенности. Первый хранит **ссылку** на массив блоков, в которых лежит продолжение файла. Если разрядности первого элемента не хватило (32 бита на ссылку могут указать лишь $512 / 4 = 128$ блоков) чтобы уместить весь файл, идем в `ic_ib[1]`, который хранит **ссылку на массив ссылок**, куда поместится гораздо больше блоков. Если опять не влезли, идем в `ic_ib[2]`. Всё это нужно, чтобы не располагать огромный файл со всем содержимым в одном месте, а с помощью ссылок распределить по файловой системе, как будет удобно.

Также стоит обратить внимание на слайде на типы файлов в нашей ФС, а также представление директории (поля не как у файла, а всего лишь информационные + набор inode'ов, которые лежат в этой директории)

(Не знаю, надо ли, но вдруг)

icommon - запись о файле

smode - права и режимы доступа

nlink - кол-во жёстких ссылок

suid - юзер

sgid - группа юзера

lsize - размер

atime (access) - время доступа

mtime (modification) - время модификации

ctime (creation) - время создания

flags - флаги

blocks - кол-во занимаемых блоков на диске

ic_gen - номер поколения

ic_shadow - теневой дескриптор для ACL

uid, gid - 32-разрядные версии suid и sgid

Типы файлов в Unix перечислены на слайде.

direct - запись о каталоге

d_ino - указатель на inode

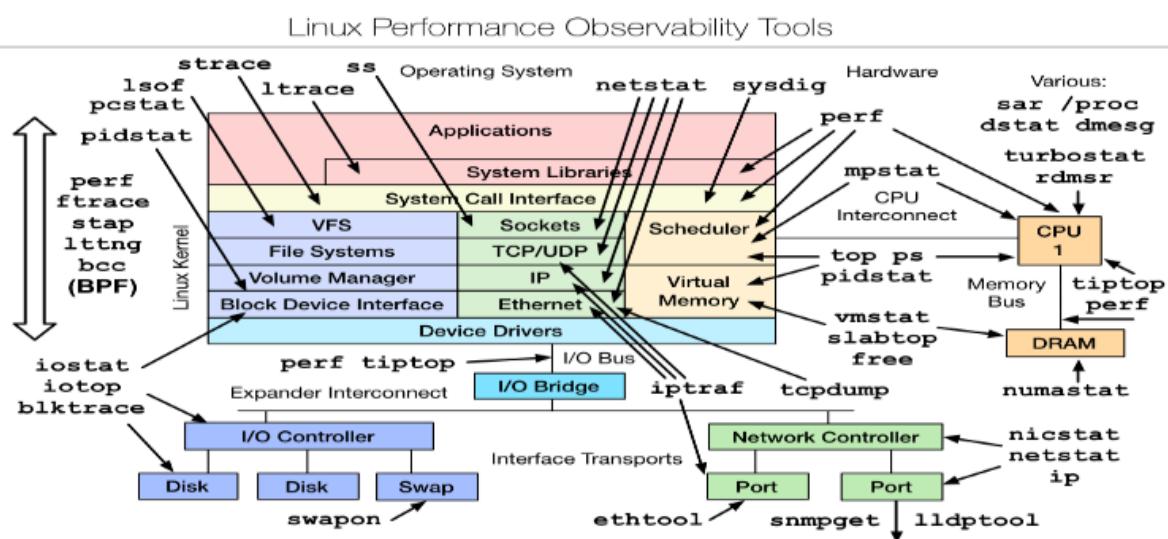
d_reclen - размер записи

d_namlen - размер имени

d_name[256] - имя

70. Linux: стандартные средства для наблюдения счетчиков ядра.

Обязательная картинка:



Хотите или не хотите - линуха собирает информацию. Большинство утилит (которые неинтрузивные) просто на основе собранных данных показывают красивый вывод.



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 (-l), 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

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



sar - наблюдение за счетчиками ядра, получение информации о почти всех подсистемах Linux.

Мониторинг прерываний, электропитания, работы сетевых устройств и др. в зависимости от ключей.

Crontab - автоматический сбор необх. информации за опред. промежуток времени в бинарные файлы.

-q - очередь процессов, 1 1 - кол-во измерений и интервал в секундах между ними (первый параметр опционален)

ldavg - load average

runq - run queue



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!

ps - выдает список процессов и некоторые доп. данные (имя команды, время работы, TTY, PID и т.д.), обычно используется в комбинации с grep для убийства процессов.

top - аналогична ps, но работает интерактивно (по умолчанию обновляется раз в секунду).

Утилиты на “stat” выдают заданное кол-во измерений за заданный промежуток времени.

Утилиты на “top” выдают список, обновляющий через определённое время.

Остальные утилиты специфические, чтобы разобраться в особенностях их работы, нужно их запускать и читать мануалы.

Утилиты также различаются по интерактивности (top работает интерактивно, sar - задаёшь параметры, и он работает до остановки) и правам (многие работают только под rootом).

Подробнее [тут](#).

71. Linux: файловая система /proc.



- Виртуальная файловая система, содержащая файлы статистики и управляющая модулями ядра
- `find /proc |wc -l`
 - 398401 (over 9000!)
- Для начала:
 - `/proc/cpuinfo` - информация о процессоре (модель, семейство, размер кэша и т.д.)
 - `/proc/meminfo` - информация о памяти, размере области подкачки и т.д.
 - `/proc/mounts` - список смонтированных файловых систем.
 - `/proc/devices` - список устройств.
 - `/proc/filesystems` - поддерживаемые файловые системы.
 - `/proc/modules` - список загружаемых модулей.
 - `/proc/version` - версия ядра.
 - `/proc/cmdline` - список параметров, передаваемых ядру при загрузке.

Более подробно смотри:
kernel.org -- ver-- Documentation/filesystems/proc.rst

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/filesystems/proc.rst

task specific:

File	Content
<code>clear_refs</code>	Clears page referenced bits shown in smaps output
<code>cmdline</code>	Command line arguments
<code>cpu</code>	Current and last cpu in which it was executed (2.4)(smp)
<code>cwd</code>	Link to the current working directory
<code>environ</code>	Values of environment variables
<code>exe</code>	Link to the executable of this process
<code>fd</code>	Directory, which contains all file descriptors
<code>maps</code>	Memory maps to executables and library files (2.4)
<code>mem</code>	Memory held by this process
<code>root</code>	Link to the root directory of this process
<code>stat</code>	Process status
<code>statm</code>	Process memory status information
<code>status</code>	Process status in human readable form
<code>wchan</code>	Present with CONFIG_KALLSYMS=y: it shows the kernel function symbol the task is blocked in - or "0" if not blocked.
<code>pagetable</code>	Page table
<code>stack</code>	Report full stack trace, enable via CONFIG_STACKTRACE
<code>smaps</code>	An extension based on maps, showing the memory consumption of each mapping and flags associated with it
<code>smaps_rollup</code>	Accumulated smaps stats for all mappings of the process. This can be derived from smaps, but is faster and more convenient
<code>numa_maps</code>	An extension based on maps, showing the memory locality and binding policy as well as mem usage (in pages) of each mapping.

/proc - каталог, представляющий собой (далее как на слайде).

Системные свойства представляются в виде файлов.

Содержимое каталога может меняться от версии к версии.

Запись в некоторые файлы может приводить к определенным событиям (через запись 'g' в /proc/sysrq-trigger можно вызвать отладчик ядра, подробнее [тут](#)).



/proc (2)

- Можно получить много полезной информации о выполняющихся процессах

```
serge@ra:~$ echo $$  
13318  
serge@ra:~$ cd /proc/13318  
serge@ra:/proc/13318$ ls -F  
arch_status      cpuset    loginuid    numa_maps      sched      status  
attr/            cwd@     map_files/   oom_adj       schedstat   syscall  
autogroup        environ   maps         oom_score    sessionid   task/  
auxv            exe@     mem          oom_score_adj setgroups   timers  
cgroup           fd/      mountinfo   pagemap      smaps      timerslack_ns  
clear_refs       fdinfo/   mounts      patch_state  smaps_rollback uid_map  
cmdline          gid_map  mountstats personality  stack      wchan  
comm             io       net/        projid_map   stat       statm  
coredump_filter  limits   ns/         root@  
serge@ra:/proc/13318$ cat wchan  
do_wait
```

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

Информация о bash; \$\$ - код текущего процесса.

fd - файловые дескрипторы всех открытых процессом файлов.

wchan (waiting channel) - функция ядра, где находится спящий процесс

shell спит, потому что ждёт ввод с терминала.

72. Linux: трассировщики системных вызовов и библиотек.



Трассировщики

- Трассировка системных вызовов: strace
- Трассировка вызовов библиотек: ltrace
- Трассировка lock -офф(ф): bpftrace

Инtrузивный.

1. strace (стравс) — трассировка системных вызовов
 2. ltrace — трассировка библиотек

<https://habr.com/ru/company/badoo/blog/493856/> - тут как она работает.

Делаем fork, выполняем системный вызов ptrace(PTRACE_TRACEME): процесс потомок ожидает, что процесс родитель будет его отслеживать, выполняем exec, чтобы запустить нашу программу.

```
/* Child here */

/* A traced mode has to be enabled. A parent
 * to happen. */

ptrace(PTRACE_TRACEME, 0, NULL, NULL);

/* Replace itself with a program to be run */

execvp(argv[1], argv + 1);

err(EXIT_FAILURE, "exec");
```

Процесс-родитель должен теперь вызвать `wait(2)` в дочернем процессе, то есть убедиться, что переключение в режим трассировки произошло

Теперь вызов `ptrace(PTRACE_SYSCALL)` гарантирует, что последующий `wait` родителя завершится либо перед исполнением системного вызова, либо сразу после его завершения. Нам же достаточно вызвать команду `ptrace(PTRACE_GETREGS)`, чтобы получить состояния регистров:

Получается что-то типа такого:

`ptrace(PTRACE_SYSCALL, child_pid, NULL, NULL)` – stop right before syscall

```
waitpid(child_pid, NULL, 0) – wait child  
ptrace(PTRACE_GETREGS, child_pid, NULL, &registers) – get registers  
– do something  
ptrace(PTRACE_SYSCALL, child, NULL, NULL) – exec syscall  
ptrace(PTRACE_GETREGS, child_pid, NULL, &registers) – get returned registers
```

73. Linux: Профилировщик perf и FlameGraph.

Профилировщик perf - это инструмент анализа производительности в Linux, позволяющий собирать с аппаратных счетчиков системную информацию, указанную ему пользователем, с помощью стека возвратов.

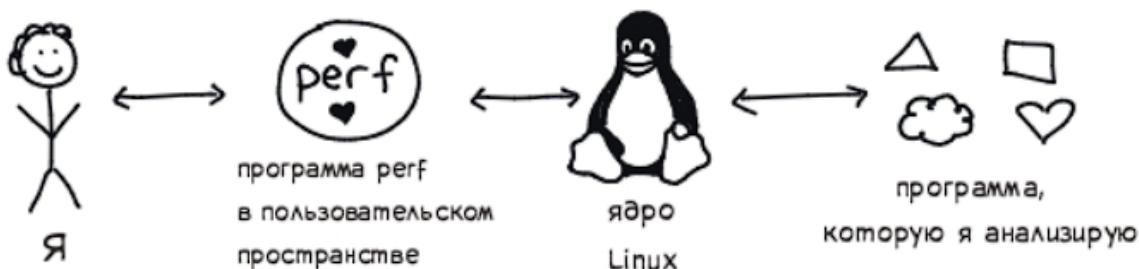
Сначала вызывается perf record. Далее указывается команда или часть работающей системы, которую необходимо профилировать. Perf mmap'ит страницы, куда будет инфа складываться. Потом ядро ему в эти странички данные складывает и готово. После этого создаётся файл perf.data, куда счётчики записывают результат. perf report и perf script выводят соответственно профиль и трассировку на консоль в читаемом текстовом формате.

Из второго слайда понятно, что perf - это “швейцарский нож”, позволяющий изучать разную статистику по конкретной работающей команде, в том числе в момент ее запуска, планировщик задач, различные локи, счётчики ядра и многое другое. Помимо общесистемных штук может рассказать многое о самом приложении.

FlameGraph - это утилита построения красивых флеймграфов (т.е. графиков времени выполнения и вызовов функций) на основе результатов работы perf или dtrace.

perf list - можно посмотреть события (events):

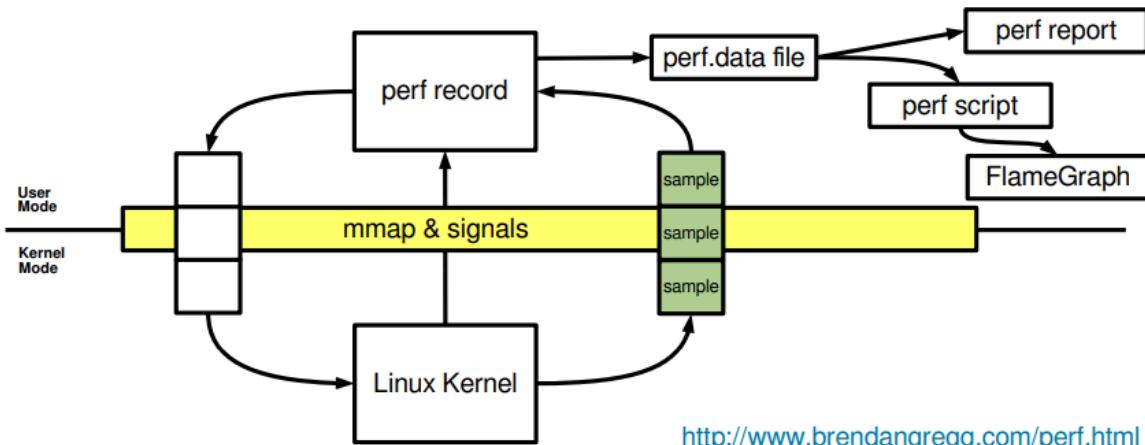
Software events, hardware events, raw hardware events, tracepoint events (sched_process_fork/wait/exec) и так далее.



mega крутой комикс про perf
<https://www.brendangregg.com/perf.html>



Профиляровщик perf (perf_events)



Операционные системы. Часть 1.

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Профиляровщик perf (user, kernel, h/w params)

- usage: perf [-version] [-help] [OPTIONS] COMMAND [ARGS]
- The most commonly used perf commands are:

<ul style="list-style-type: none"> • bench General framework for benchmark suites • c2c Shared Data C2C/HITM Analyzer. • config Get and set variables in a configuration file. • data Data file related processing • diff Read perf.data files and display the differential profile • evlist List the event names in a perf.data file • ftrace simple wrapper for kernel's ftrace functionality • kallsyms Searches running kernel for symbols • kmem Tool to trace/measure kernel memory properties • list List all symbolic event types • lock Analyze lock events • mem Profile memory accesses • record Run a command and record its profile into perf.data 	<ul style="list-style-type: none"> • report Read perf.data and display the profile • sched Tool to trace/measure scheduler properties (latencies) • script Read perf.data and display trace output • stat Gather performance counter statistics on command • timechart Tool to visualize total system behavior • top System profiling tool. • probe Define new dynamic tracepoints • trace strace inspired tool
--	---

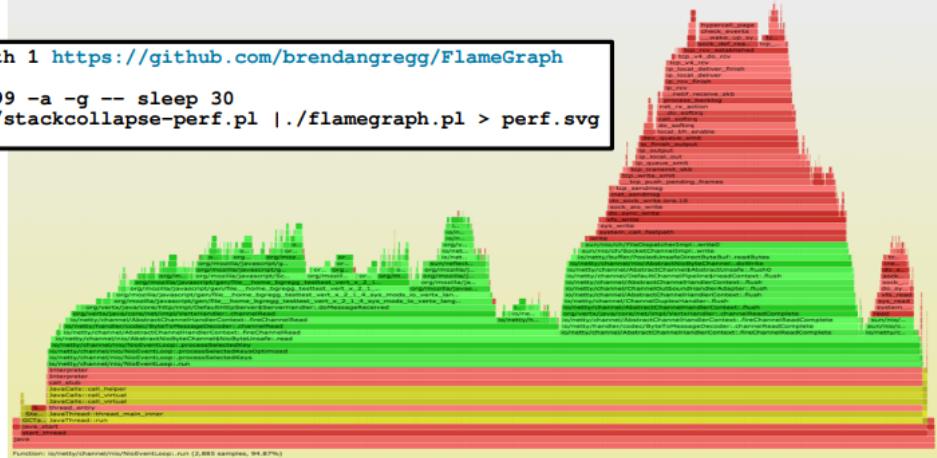


FlameGraph

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

Brendan's patched OpenJDK, Mixed Mode CPU FlameGraph: vert.x

```
git clone --depth 1 https://github.com/brendangregg/FlameGraph
cd FlameGraph
perf record -F 99 -a -g -- sleep 30
perf script | ./stackcollapse-perf.pl | ./flamegraph.pl > perf.svg
```

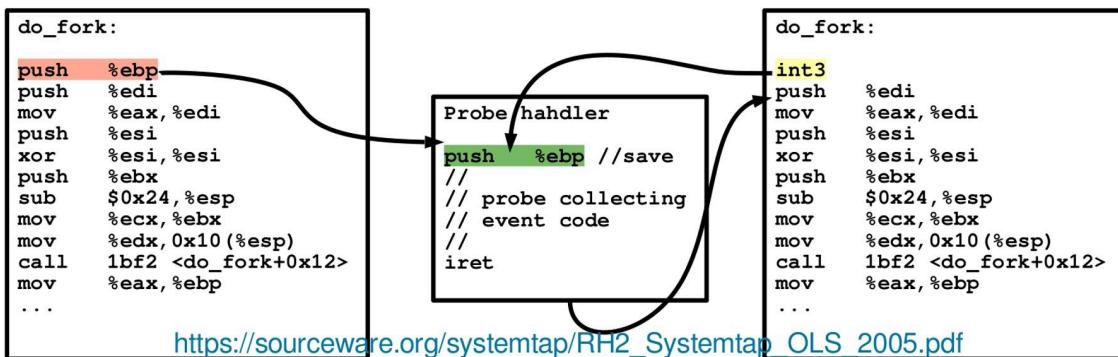


74. Linux: SystemTap.



System tap

- Средства сбора статистики по kprobes и uprobes
- «Минимальное» воздействие на систему



Операционные системы. Часть 1. Обзор операционных систем
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

SystemTap - скриптовый язык программирования и средство сбора данных (instrumentation) о подсистемах ядра и пользователю.

Probe - событие, по которому собирается статистика, и соответствующий обработчик; kprobe (kernel) на уровне ядра, uprobe - юзера.

Пример на слайде - ядерная функция `do_fork()`. stap сохраняет первую инструкцию в обработчик, а в самой функции заменяет её на прерывание; таким образом, выполняется первая команда, обработчик, а потом остальные инструкции.

Если функция вызывается прерываниями высокой частоты (существуют частые системные прерывания), “проба” будет оказывать существенное влияние на работу системы, но в общем случае воздействие stap минимально, потому что количество кода мало и вызывается он посреди функции.



Использование SystemTap

- Скриптовый язык, похожий на AWK
- `probe <event> { handler }`
 - Event может быть: syscall.function, process.statement, timer.ms, begin, end, (tapset) aliases
 - Handler: управляющие структуры, переменные (числа, строки), ассоциативные массивы или статистические агрегаторы
 - Вспомогательные функции: log, printf, gettimeofday, pid
- Множество готовых скриптов для анализа

<https://sourceware.org/systemtap/examples/keyword-index.html>

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



79

В состав SystemTap входит AWK-подобный скриптовый ЯП.

Синтаксис описания проб - `probe <event> { handler }`.

event - события под наблюдением скрипта.

Примеры:

- системный вызов - `syscall.open`;
- событие таймера (с периодичностью) - `timer.ms (10)`;
- вызов заданной строки программы - `process("a.out").statement("*@main.c:200")`;
- `begin` - вызывается в начале сессии stap, может юзаться для установки начальных значений переменных скрипта;
- `end` - в конце сессии, обычно юзается для вывода статистики.

Handler - обработчик пробы, где указываются действия, выполняемые при наступлении события (например, наращивание счётчиков).

ЯП для SystemTap содержит Си-подобные управляющие структуры (условия, циклы), переменные (числа, строки), массивы (в т.ч. ассоциативные, т.е. словари) и статистические агрегаторы (`$count`, `$avg`, `$sum`).

Вспомогательные функции: `log`, `printf`, `gettimeofday`, `pid`, ...

Разработчиками stap было написано множество скриптов для мониторинга приложений, которые можно использовать в готовом виде или в качестве шаблонов.



stap пример

```
#!/usr/bin/env stap
global opens, reads, writes, totals

probe begin {
    printf("starting probe\n")
}

probe syscall.open {
    opens[execname()] <<< 1
}

probe syscall.read.return {
    count = retval
    if (count >= 0) {
        reads[execname()] <<< count
        totals[execname()] <<< count
    }
}

probe syscall.write.return {
    count = retval
    if (count >= 0) {
        writes[execname()] <<< count
        totals[execname()] <<< count
    }
}

probe end {
    printf("\n%16s %8s %8s %8s %8s %8s %8s %8s\n",
        "", "", "", "read", "read", "", "write", "write")
    printf("%16s %8s %8s %8s %8s %8s %8s %8s\n",
        "name", "open", "read", "KB tot", "B avg",
        "write", "KB tot", "B avg")
    # sort by total io
    foreach (name in totals @sum- limit 20) {
        printf("%16s %8d %8d %8d %8d %8d %8d %8d\n",
            name, @count(opens[name]),
            @count(reads[name]),
            (@count(reads[name]) ? @sum(reads[name])>>10 : 0),
            (@count(reads[name]) ? @avg(reads[name]) : 0),
            @count(writes[name]),
            (@count(writes[name]) ? @sum(writes[name])>>10 : 0),
            (@count(writes[name]) ? @avg(writes[name]) : 0))
    }
}
```

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Пример работоспособного скрипта представлен на слайде.

Начало скрипта - глобальные переменные, где будет собираться статистика.

begin срабатывает при старте скрипта.

После компиляции кода (а процесс этот достаточно многоступенчатый и долгий) в ядро внедряется специальный модуль, полученный из программы.

Проба на системный вызов open - ассоциативный массив opens, ключ execname() - имя исполняемого файла. Подсчёт количества открываемых файлов.

probe syscall.read.return - на каждый возврат из системного вызова read, если больше 0, сохраняем возвр. значение (кол-во прочит. байт) в ассоц. массивы.

Аналогично для write.

end - вывод статистики на экран (здесь - 18 самых загруженных IO программ в ОС)



stap запуск примера

serge@ra:/tmp\$ sudo stap file.stap								
name	open	read	KB tot	read B avg	write	KB tot	write B avg	
localStorage DB	0	0	0	0	7	96	14053	
IndexedDB #1923	0	8	25	3212	6	8	1375	
gnome-shell	0	166	13	86	133	1	8	
Web Content	0	7581	7	1	4769	4	1	
irqbalance	0	18	4	256	1	0	8	
zoom	0	7	4	635	0	0	0	
pool-gnome-shel	0	16	3	196	40	0	8	
GraphRunner	0	0	0	0	1924	1	1	
MainThread	0	958	0	1	519	0	1	
InputThread	0	19	1	72	19	0	1	
acpid	0	57	1	24	0	0	0	
gdbus	0	48	0	8	95	0	8	
Timer	0	0	0	0	969	0	1	
Chrome_~dThread	0	449	0	1	415	0	1	
Xorg	0	39	0	16	0	0	0	
Gecko_IOThread	0	440	0	1	143	0	1	
IPDL Background	0	0	0	0	434	0	1	
threaded-ml	0	32	0	5	72	0	2	

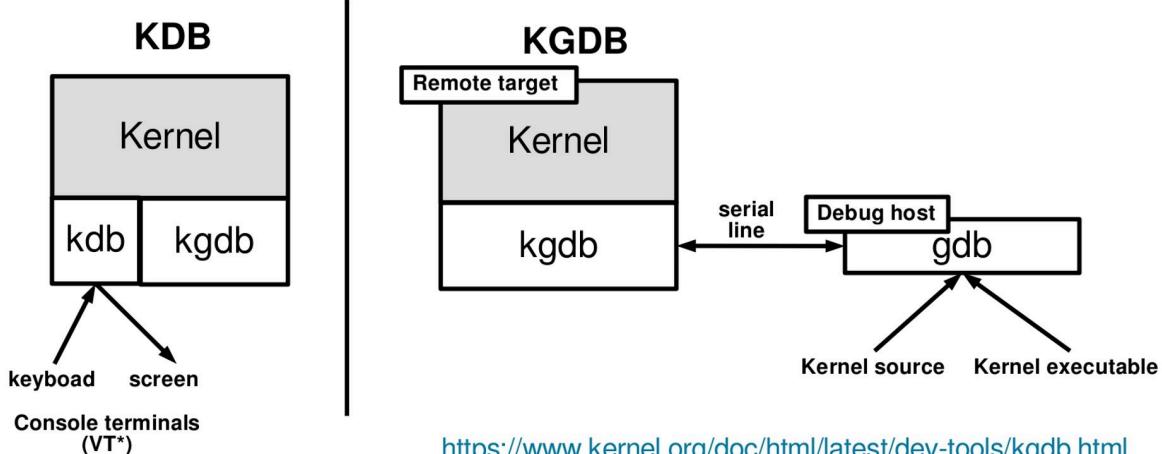
Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

75. Linux: Отладчик ядра.



Отладчик ядра



<https://www.kernel.org/doc/html/latest/dev-tools/kgdb.html>

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

Отладчик ядра - ПО, работающее на уровне ядра ОС и позволяющее отлаживать ядро и вредоносный код, устранять BSOD(kernel panic), искать уязвимости, писать эксплойты и т.д.

Две конфигурации отладчика:

- 1) Локальный отладчик - запуск мини-оболочки с терминала.

- 2) Удаленный отладчик - запускается поserialному порту (физическому или виртуальному). Отладка с одной ОС в другой (К.О.) с помощью gdb, создающего команды для управления удаленной системой. (kgdboc=<serial_device>,...)

Вывод полного списка команд с помощью help или знака вопроса.

Простейшие команды, создающие “распечатки памяти”, позволяющие посмотреть, что находится в памяти и буфере.

Вызов KDB

- Загрузить ядро с параметром kgdboc=kbd (через /etc/default/grub или меню загрузки)

```
root@ra:~# cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-5.4.0-47-generic
root=UUID=cb68f380-c1a3-4757-a4f2-73b76b9e0934
ro ipv6.disable=1 kgdboc=kbd
```

- Запустить с виртуального терминала триггер переключения
Система встанет!

```
root@ra:~# echo g > /proc/sysrq-trigger
Entering kdb ...
[1]kdb> ?
[1]kdb> go
```

- 1) Добавить kgdboc=kbd в опции загрузки или дописать в переменную GRUB_CMDLINE_LINUX файла /etc/default/grub (только sudo; потом надо sudo update-grub).
- 2) Перезагрузить компьютер.
- 3) Перейти в любой виртуальный терминал (Ctrl+Alt+F3 или sudo chvt <num>).
- 4) Послать системе запрос на переключение режимов (Alt+SysRq(где PrtSc)+g или echo g > /proc/sysrq-trigger, только под rootом).
- 5) Система перейдет в отладчик и остановит все процессы, пока не будет введено g или go. В это время можно изучать память структур ядра.

P.S. Ясное дело, SysRq (/proc/sys/kernel/sysrq) и KDB/KGDB (конфиг ядра) должны быть активны, но это уже чутка за рамками лекционного материала.

76. Windows: стандартные отладочные средства.



Отладочные средства Windows

- Встроенные стандартные средства
- Windows SDK
 - Отладчики, множество утилит, поддерживающих сборку приложений
 - DTrace on Windows
<https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/dtrace>
- SysInternals
- Сторонние средства
 - <https://checkpanel.com/resources/windows-server-performance-monitoring-tools>

Windows SDK - средство разработки ПО на Windows, содержащее много средств наблюдения за производительностью ОС.

DTrace - средство авторства Sun Microsystems, позволяющее вычислять счётчики ядра, контролировать вход и выход из функций, и создавать информативные графики и гистограммы для визуального анализа процесса. Microsoft его портировала на Windows.

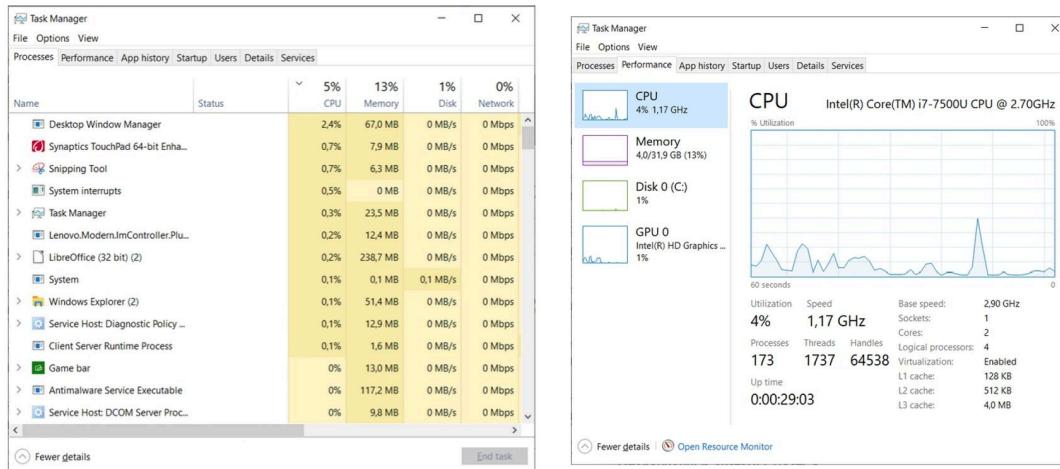
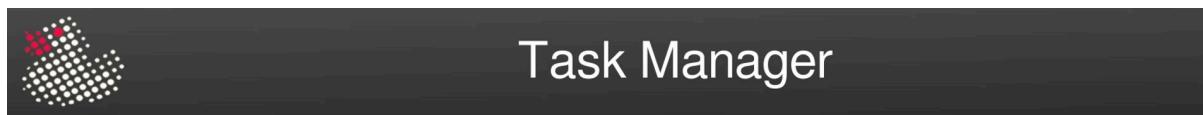
SysInternals - набор средств для получения информации о системе, позволяющий нивелировать меньшую функциональность командной строки Windows по сравнению с Unix (подробнее в пределах нескольких следующих слайдов).

И многие другие...



Стандартные средства Windows

- Находятся в ControlPanel
- В большинстве случаев — средства изменения конфигурации системы и наблюдения за ее поведением
- Control Panel → System and Security → Administrative Tools
- Disk Cleanup, Performance Monitor, Resource Monitor, Registry Editor, Services, System Configuration,



Операционные системы. Часть 1. Обзор операционных систем

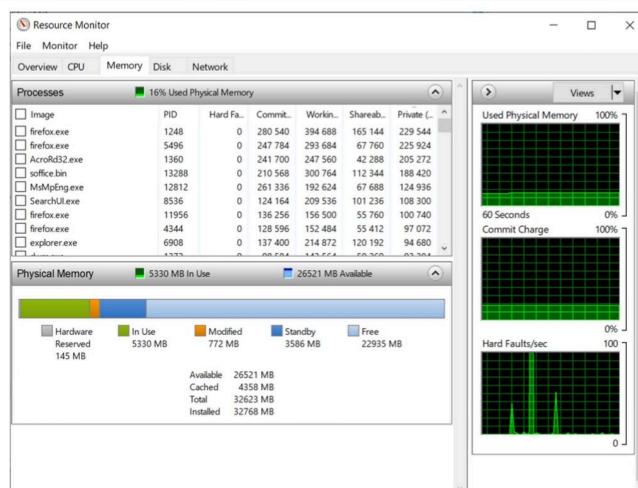
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Вывод списка процессов, сортируемых по потребляемым ресурсам (CPU, RAM, диски, GPU, сеть)

Снятие процессов с исполнения

Визуализация потребления ресурсов во времени в виде графиков

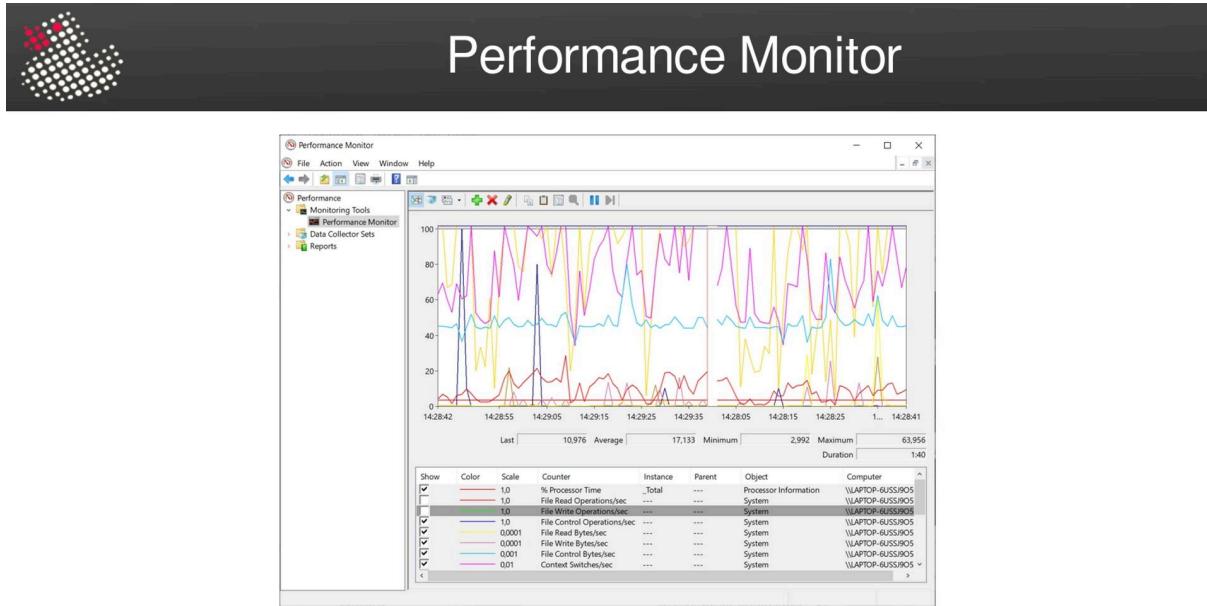


Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Тоже показывает потребление ресурсов, но в слегка ином виде, используется реже.



Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Огромный список контролируемых параметров (всё, что доступно в интерфейсе ядра и логируемо).

Не очень удобен - при большом кол-ве параметров на мониторе происходит неразбериха.

Можно собирать данные за определённое время и потом проводить их анализ.

Top SysInternals utils

- PsList and PsKill — просмотр и остановка процессов (в том числе и удаленно)
- Process Explorer — просмотр ресурсов процесса, замена Task Manager
- Process Monitor — просмотр связанных с процессом ресурсов реестра
- Autoruns — поиск автозапускаемых программ
- Contig — дефрагментирует конкретный файл
- PSFile — позволяет показать открытые файлы, в том числе и удаленно
- MoveFile — перемещает заблокированные файлы во время перезагрузки.
- Sync — синхронизация файловой системы
- TCPview — информация о открытых сетевых соединениях
- SDelete — удалить файлы и папки без возможности восстановления

SysInternals - мусорка с кучей разных программ для отладки винды, часть из них копирует встроенные линуксовые утилитки, некоторые имеют графический, некоторые консольный интерфейс, а некоторые вообще служебного толка (например livekd).



Отладчик WinDbg и KD

- Требуют загрузки символьной информации о ядре
- Без livekd (SysInternals) требуют перезагрузки ядра в отладочном режиме
- Требуют привилегий администратора

```
set _NT_SYMBOL_PATH=srv*c:\symbols*http://msdl.microsoft.com/download/symbols  
C:\Program Files (x86)\SysinternalsSuite>livekd64.exe -w -k "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe"
```

WinDbg (дамп) + livekd (ядро в режим отладки без перезапуска) = можно поглядеть дамп памяти винды в символьном виде.

77. Windows: утилиты SysInternals



SysInternals

- Множество скриптов и программ для получения информации о системе
- Автор — Марк Руссинович, в настоящее время сотрудник Microsoft (соавтор книги Windows Internals)
- Отдельно загружается и устанавливается с сайта Microsoft

<https://docs.microsoft.com/ru-ru/sysinternals/>



Top SysInternals utils

- PsList and PsKill — просмотр и остановка процессов (в том числе и удаленно)
- Process Explorer — просмотр ресурсов процесса, замена Task Manager
- Process Monitor — просмотр связанных с процессом ресурсов реестра
- Autoruns — поиск автозапускаемых программ
- Contig — дефрагментирует конкретный файл
- PSFile — позволяет показать открытые файлы, в том числе и удаленно
- MoveFile — перемещает заблокированные файлы во время перезагрузки.
- Sync — синхронизация файловой системы
- TCPview — информация о открытых сетевых соединениях
- SDelete — удалить файлы и папки без возможности восстановления

Операционные системы. Часть 1. Обзор операционных систем

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

SysInternals позволяет нивелировать меньшую функциональность командной строки Windows по сравнению с Unix. Например, он содержит утилиты для получения из командной строки информации о запущенных процессах и “убийства”, которые привычны для юниксайдов, но в виндовской CMD отсутствуют. При этом часть средств является графической.

Эти утилиты позволяют наблюдать все процессы на удалённых машинах, к которым в Windows разрешён доступ, что значительно повышает их полезность.

Process Explorer и Process Monitor по сравнению с Task Manager “лезут в процесс” гораздо глубже, поэтому предоставляют пользователю больше информации системного уровня.

С помощью многих из средств SysInternals возможно фиксировать “тロяны” и выгружать их из памяти, потому что первые возникли из решений, направленных на борьбу с последними. Например, MoveFile позволяет осуществить перенос файла, заблокированного “тロяном”.

Autoruns выявляет и помогает удалить автозапускаемые программы, что также помогает в борьбе с вирусами.

Процедура Config осуществляет дефрагментацию указанного файла, при этом его куски располагаются на диске последовательно, и это позволяет Windows работать с ним быстрее.

TCPView показывает открытые соединения в графически красивом виде и “кто куда ходит”. Однако, когда запущен браузер, разобраться во множестве запущенных сетевых соединений достаточно сложно, поскольку практически каждая запущенная web-страница “тянет” за собой много связанных ссылок, в том числе и рекламного характера.

Livekd без перезагрузки Windows (просто запуская для этого необходимое) переводит систему в режим отладки и позволяет читать информацию непосредственно из сегмента ядра.

78. Windows: отладчики WinDbg и KD



Отладчик WinDbg и KD

- Требуют загрузки символьной информации о ядре
- Без livekd (SysInternals) требуют перезагрузки ядра в отладочном режиме
- Требуют привилегий администратора

```
set _NT_SYMBOL_PATH=srv*c:\symbols*http://msdl.microsoft.com/download/symbols  
C:\Program Files (x86)\SysinternalsSuite>livekd64.exe -w -k "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe"
```

Операционные системы. Часть 1. Обзор операционных систем
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Для запуска WinDbg по умолчанию требуется перезагрузка в отладочном режиме. Утилита livekd из SysInternals переводит систему в режим отладки без перезагрузки (просто перезапуская все для этого необходимое) переводит систему в режим отладки и позволяет читать информацию непосредственно из сегмента ядра. Информация в ядре представляется в бинарном формате, поэтому для получения чего-то осмыслиенного требуется получить от Microsoft символьную информацию (информацию о переменных ядра) о ядре, индивидуальную для каждой сборки, то есть при обновлении все придется перекачивать. Тогда необходимо перед запуском отладчика указать путь, где лежит эта информация.

```

C:\Windows\System32\notepad.exe - WinDbg Version 10.0.21306.1007 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: C:\Windows\System32\notepad.exe

***** Path validation summary *****
Response           Time (ms)    Location
Deferred          0            srv*
Symbol search path is: srv*
Executable search path is:
ModLoad: 00007ff6`6e760000 00007ff6`6e798000  notepad.exe
ModLoad: 00007ff8`08d00000 00007ff8`08fa5000  ntdll.dll
ModLoad: 00007ff8`07220000 00007ff8`072dd000  C:\WINDOWS\System32\KERNEL32.DLL
ModLoad: 00007ff8`06b40000 00007ff8`06e08000  C:\WINDOWS\System32\KERNELBASE.dll
ModLoad: 00007ff8`07c00000 00007ff8`07c2a000  C:\WINDOWS\System32\GDI32.dll
ModLoad: 00007ff8`06ab0000 00007ff8`06ad2000  C:\WINDOWS\System32\win32u.dll
ModLoad: 00007ff8`066a0000 00007ff8`067ab000  C:\WINDOWS\System32\gdi32full.dll
ModLoad: 00007ff8`06a10000 00007ff8`06aad000  C:\WINDOWS\System32\msvcp_win.dll
ModLoad: 00007ff8`067b0000 00007ff8`068b0000  C:\WINDOWS\System32\ucrtbase.dll
ModLoad: 00007ff8`07820000 00007ff8`079c0000  C:\WINDOWS\System32\USER32.dll
ModLoad: 00007ff8`07420000 00007ff8`07775000  C:\WINDOWS\System32\combase.dll
ModLoad: 00007ff8`08480000 00007ff8`085ab000  C:\WINDOWS\System32\RPCRT4.dll
ModLoad: 00007ff8`08c40000 00007ff8`08cee000  C:\WINDOWS\System32\shcore.dll
ModLoad: 00007ff8`085b0000 00007ff8`0864e000  C:\WINDOWS\System32\msvcrt.dll
ModLoad: 00007fff`f8580000 00007fff`f881a000  C:\WINDOWS\WinSxS\amd64_microsoft.windows.common-cont
(5500.34d8): Break instruction exception - code 80000003 (first chance)
ntdll!LdrpDoDebuggerBreak+0x30:
00007ff8`08e80570 cc          int     3

```

79. Аппаратная поддержка взаимных исключений.

По лекциям:

- Запрет прерываний на однопроцессорных системах
 - DI; Критический участок; EI
- Атомарные инструкции TAS, CAS, CMPXCHG, ...

```

spin_acquire_lock: // locked = 1, released = 0
    movl $1, %ebx
    movl lock_var_addr, %ecx
loop: pause //rep nop
    movl (%ecx), %eax //avoid unness. lock
    test %eax, %eax
    jnz loop
    lock cmpxchg %ebx, (%ecx) // %eax = 0
    jnz loop
    ret

spin_unlock:
    mfence
    movl$0, (lock_var_addr)
    ret

```

/* Atomic compare-and-exchange: */
Compare eax with memory (%ecx)
if equal
 load %ebx in memory (%ecx), ZF=1
else
 load memory in %eax, ZF=0

pause - hint CPU we are spinning
mfence - load/store barriers

<https://www.cs.virginia.edu/~cr4bd/4414/F2018/slides/20180925--slides-1up.pdf>

TAS - test and set, CAS - compare and swap, CMPCHG - compare and exchange

mfence - все инструкции чтения/записи завершат работу к этому моменту (все write buffers of cpus должны освободиться).

lock - блокируется доступ к линии кеша (или ячейке, но тут точно не помню). соответственно время будет гораздо большее.

По стоплингу:

Отключение прерываний.

Работает только на **однопроцессорных** машинах. К тому же это сильно влияет на производительность.

В ядре может быть что-то наподобие:

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

Атомарные инструкции.

1. Compare&swap. Também может возвращать булевое значение.

```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
    return oldval;
}
```

Пример программы с compare&swap.

```
/* program  mutual exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
        while (compare_and_swap(bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main() {
    bolt = 0;
    parbegin (P(1), P(2), ... , P(n));
}
```

2. Exchange

```
void exchange (int *register, int *memory)
{
    int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
}
```

Пример программы с exchange

```

/* program mutual exclusion */
int const n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true)
        int keyi = 1;
        do exchange (&keyi, &bolt)
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main() {
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}

```

Преимущества атомарных инструкций:

1. Работают на многопроцессорных машинах.
2. Просты в реализации (памятник аппаратчикам).
3. С их помощью можно поддерживать несколько критических секций. Для каждой секции нужна лишь своя переменная.

Недостатки атомарных операций:

1. Применяется busy waiting, а значит простой процессора.
2. Возможно голодание процессов потому что блокирующее.
3. Возможны deadlock-и. Пример: Процесс P1 выполнил compare&swap и вошел в критическую секцию, но после этого был прерван процессом P2 с более высоким приоритетом. Теперь P2 не может использовать тот же ресурс, что и P1.

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

В главе 5.1 рассматривается программный подход для взаимных исключений, который может быть реализован как в однопроцессорной, так и в многопроцессорной системах с общей основной памятью. Обычно такой подход предполагает элементарные взаимоисключения на уровне доступа к памяти.

Алгоритм Деккера (который все пытался):

<pre>/* Процесс 0 */ : : while (turn != 0) /* Ничего не делаем */; /* Критический участок */; turn = 1; :</pre>	<pre>/* Процесс 1 */ : : while (turn != 1) /* Ничего не делаем */; /* Критический участок */; turn = 0; :</pre>
---	---

a) Первая попытка

<pre>/* Процесс 0 */ : : while (flag[1]) /* Ничего не делаем */; flag[0] = true; /* Критический участок */; flag[0] = false; :</pre>	<pre>/* Процесс 1 */ : : while (flag[0]) /* Ничего не делаем */; flag[1] = true; /* Критический участок */; flag[1] = false; :</pre>
--	--

б) Вторая попытка

Первая попытка:

Как упоминалось ранее, любая попытка взаимного исключения должна опираться на некий фундаментальный механизм исключений аппаратного обеспечения. Наиболее общим механизмом может служить ограничений, что к некоторой ячейке памяти в определенный момент времени может осуществляться только одно обращение.

Воспользовавшись этим ограничением, зарезервируем глобальную ячейку памяти, которую назовем turn. Перед выполнением критического участка, сначала будем проверять значением ячейки памяти turn. Если значение равно номеру процесса, то процесс может войти в крит участок, в противном случае ожидаем (пережидание занятости).

Недостатки:

- При входе в критический участок процессы должны чередоваться.
- В случае сбоя одного из процессов, другой процесс остается заблокированным.

Вторая попытка:

Введем вектор flag, в котором flag[0] соответствует процессу P0, а flag[1] - процессу P1. Когда процессу требуется войти в крит участок, он периодически проверяет состояние флага другого процесса до тех пор, пока тот не примет значение false, указывающее то, что другой процесс покинул крит участок. Тогда процесс немедленно устанавливает значение собственного флага, равного true и входит в крит участок. После выхода сбрасывает свой флаг.

Теперь если произойдет сбой одного из процессов вне критического участка, второй заблокирован не будет. Однако если сбой произойдет в критическом участке (или перед входом в него, но после установки флага), то другой процесс окажется заблокированным навсегда.

Также описанное решение, по сути, оказывается еще хуже предложенного ранее, поскольку даже не гарантирует взаимного исключения. Например при последовательности:

- P0: while(flag[1]) // flag[1] == false
- P1: while(flag[0]) // flag[0] == false
- P0: flag[0] = true; in critical section
- P1: flag[1] = true; in critical section

Таким образом, программа некорректна.

/* Процесс 0 */	/* Процесс 1 */
<pre>: : flag[0] = true; while (flag[1]) /* Ничего не делаем */; /* Критический участок */; flag[0] = false; :</pre>	<pre>: : flag[1] = true; while (flag[0]) /* Ничего не делаем */; /* Критический участок */; flag[1] = false; :</pre>

в) Третья попытка

/* Процесс 0 */	/* Процесс 1 */
<pre>: : flag[0] = true; while (flag[1]) { flag[0] = false; /* Задержка */ flag[0] = true; } /* Критический участок */; flag[0] = false; :</pre>	<pre>: : flag[1] = true; while (flag[0]) { flag[1] = false; /* Задержка */ flag[1] = true; } /* Критический участок */; flag[1] = false; :</pre>

г) Четвертая попытка

Третья попытка:

Чтобы избежать того, что процесс может изменить свое состояние после того, как другой процесс ознакомится с ним, будем устанавливать флаг перед проверкой.

Как и ранее, если произойдет сбой одного процесса в критическом участке, включая код установки значения флага, то второй процесс окажется заблокированным, а если вне, то не будет.

Однако третья попытка порождает еще одну проблему: если оба процесса установят значения флагов равными true до того, как один из них выполнить while, то каждый из процессов будет считать, что другой находится в критическом участке, и тем самым осуществляется взаимоблокировка.

Четвертая попытка:

В третьей попытке взаимоблокировка возникает по той причине, что каждый процесс мог добиваться своих прав на вход в критический участок и не было никакой возможности отступить назад (ну поэтому добавим костыль в {} после while).

```
P0 устанавливает значение flag[0] равным true;
P1 устанавливает значение flag[1] равным true;
P0 проверяет flag[1];
P1 проверяет flag[0];
P0 устанавливает значение flag[0] равным false;
P1 устанавливает значение flag[1] равным false;
P0 устанавливает значение flag[0] равным true;
P1 устанавливает значение flag[1] равным true.
```

осуществляем проверку в while еще раз.

Строго говоря, это не взаимоблокировка, так как любое изменение относительной скорости двух процессов разорвет замкнутый круг и позволит одному из процессов войти в критический участок. Назовем такую ситуацию **динамической взаимоблокировкой (livelock)**.

Хотя описанный выше сценарий маловероятен и вряд ли продлится сколь-нибудь долго, теоретически такая возможность имеется.

Правильное решение:

```
void P0()
{
    while(true)
    {
        flag[0] = true;
        while(flag[1])
            if (turn == 1)
            {
                flag[0] = false;
                while(turn == 1) /* Ничего не делать */;
                flag[0] = true;
            }
        /* Критический участок */;
        turn = 1;
        flag[0] = false;
        /* Остальной код */;
    }
}
```

```
void main()
{
    flag[0] = false;
    flag[1] = false;
    turn = 1;
    parbegin(P0, P1);
}
```

Добавим переменную turn из первой попытки, которая будет указывать чья очередь войти в критический участок.

Сам алгоритм: когда процесс P0 намерен войти в крит участок, он устанавливает свой флаг, равным true, а затем проверяет состояние флага процесса P1. Если он равен false, P0 может немедленно входить в критический участок; в противном случае P0 обращается к переменной turn. Если turn == 0, это означает, что сейчас - очередь процесса P0 на вход в критический участок, и P0 периодически проверяет состояние флага процесса P1. Этот процесс, в свою очередь, в некоторый момент времени обнаруживает, что сейчас не его очередь для входа в крит участок, и устанавливает свой флаг равным false, давая возможность войти процессу P0. После того как P0 выйдет из критического участка, он установит свой флаг равным false для освобождения критического участка и присвоит переменной turn значение 1 для передачи прав на вход в критический участок процессу P1.

Алгоритм Петерсона, а то Деккер решает задачу достаточно сложным путем, корректность которого не так легко доказать:

```

boolean flag [2];
int turn;
void P0()
{
    while (true)
    {
        flag [0] = true;
        turn = 1;
        while (flag [1] && turn == 1) /* Ничего не делать */;
        /* Критический раздел */;
        flag [0] = false;
        /* Остальной код */;
    }
}
void P1()
{
    while (true)
    {
        flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0) /* Ничего не делать */;
        /* Критический раздел */;
        flag [1] = false;
        /* Остальной код */;
    }
}
void main()
{
    flag [0] = false;
    flag [1] = false;
    parbegin(P0, P1);
}

```

Как и ранее, глобальная переменная flag указывает положение каждого процесса по отношению к взаимоисключению, а глобальная переменная turn разрешает конфликты одновременности.

Рассмотрим процесс P0. После того как flag[0] установлен им равным true, P1 войти в критический участок не может. Если же P1 уже находится в критическом участке, то flag[1] = true и для P0 вход в критический участок заблокирован.

Предположим, что P0 заблокирован в своем цикле while. Это означает, что flag[1] == true, turn == 1. P0 может войти в критический участок, когда либо flag[1] становится равным false, либо turn становится равным 0. Рассмотрим случаи:

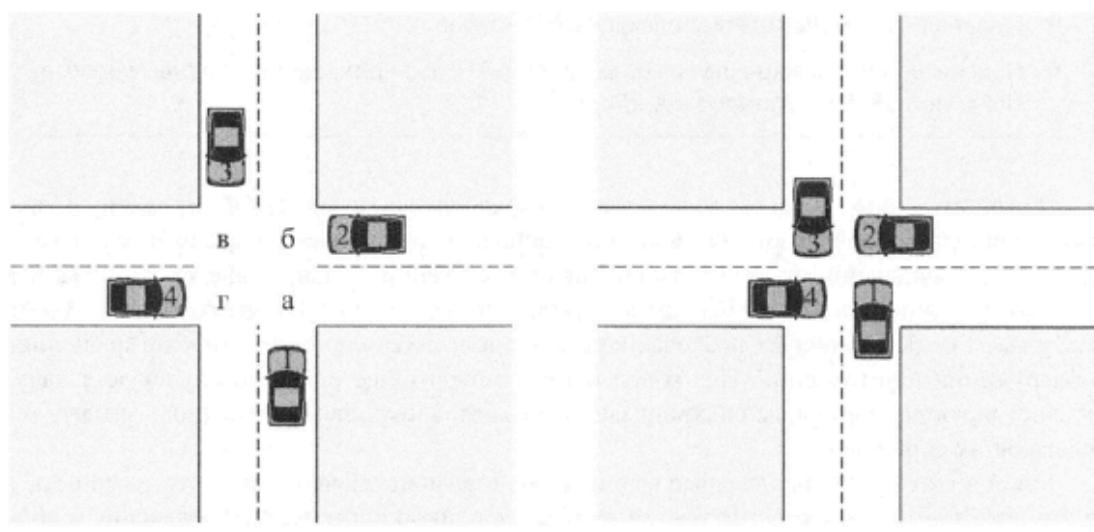
- P1 не намерен входить в критический участок. Такой случай невозможен, поскольку при этом выполнялось бы условие $flag[1] = false$.
- P1 ожидает входа в критический участок. Такой случай также невозможен, поскольку если $turn=1$, то P1 способен войти в критический участок.
- P1 циклически использует критический участок, монополизировав доступ к нему. Этого не может произойти, поскольку P1 вынужден перед каждой попыткой входа в критический участок дать возможность входа процессу P0, устанавливая значение $turn$ равным 0.

Алгоритм Петерсона легко обобщается на случай n процессов.

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

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

Обычное правило пересечения перекрестка состоит в том, что автомобиль должен уступить дорогу движущемуся справа. Это правило работает, когда перекресток пересекают два или три автомобиля. Например, если на перекрестке встретятся автомобили, движущиеся на север и на запад, то автомобиль, движущийся на север, уступит дорогу автомобилю, движущемуся на запад. Но если перекресток пересекают одновременно четыре автомобиля, каждый из которых согласно правилу воздержится от въезда на перекресток, возникнет взаимоблокировка.

Она возникнет и в том случае, если все четыре машины проигнорируют правило и осторожно въедут на перекресток, поскольку при этом каждый автомобиль захватит один ресурс (один квадрант) и останется на вечной стоянке в ожидании, когда другой автомобиль освободит следующий требующийся для пересечения перекрестка квадрант. Итак, мы опять получили взаимоблокировку.

Категории ресурсов:

- повторно используемые (reusable). Могут безопасно использовать одним процессом и не истощаться (процессор, каналы io, основная и вторичная память, периферийные устройства, а также структуры данных, такие как файлы, бд и семафоры).
- расходуемые (consumable). Могут быть созданы (произведены) или уничтожены (потреблены). Например, прерывания, сигналы, информация в буферах ввода-вывода.

Условия осуществления взаимоблокировок:

1. Взаимное исключение. 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.
4. Циклическое ожидание. Существует замкнутая цепь процессов, каждый из которых удерживает как минимум один ресурс, необходимый процессу, следующему в цепи после данного.

Первые три условия являются необходимыми, но недостаточными для выполнения взаимоблокировки. Первые три условия представляют собой, по сути, незыблевые правила, в то время как условие 4 представляет собой ситуацию, которая может осуществиться при определенной последовательности запросов и освобождений ресурсов процессом.

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

Условия осуществления взаимоблокировок:

1. Взаимное исключение. 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.
4. Циклическое ожидание. Существует замкнутая цепь процессов, каждый из которых удерживает как минимум один ресурс, необходимый процессу, следующему в цепи после данного.

Первые три условия являются необходимыми, но недостаточными для выполнения взаимоблокировки. Первые три условия представляют собой, по сути, незыблемые правила, в то время как условие 4 представляет собой ситуацию, которая может осуществиться при определенной последовательности запросов и освобождений ресурсов процессом.

Подходы решения проблем взаимоблокировки:

- Предотвращение. Не допускаем возможности возникновения взаимоблокировки.
- Устранение. Допускает наличие трех необходимых условий возникновения взаимоблокировок, но мы принимаем меры к тому, чтобы ситуация взаимоблокировок не могла быть достигнута. Решение о том, способен ли текущий ресурс в случае его удовлетворения привести к возникновению взаимоблокировки, принимает динамически (следовательно, необходимо знать, какие ресурсы потребуются в дальнейшем).
- Обнаружение. Определяет, имеется ли уже в данный момент блокировка.

Предотвращение:

- Косвенный метод. Предотвращение одного из первых трех условий возникновения взаимоблокировки.
- Прямой метод. Предотвращает циклическое ожидание.

Взаимоисключение: в общем случае избежать невозможно. Некоторые ресурсы, такие как файлы, могут поддерживать множественный доступ для записи, но даже в этом случае, если право записи в файл требуется нескольким процессам одновременно, то возможно возникновение взаимоблокировки.

Удержание: можно избежать, потребовав, чтобы процесс запрашивал все необходимые ресурсы одновременно, и блокировать процесс до тех пор, пока такой запрос не сможет быть выполнен полностью сразу.

Минусы:

- процесс может длительное время ожидать ресурсов, хотя мог работать только с частью из них
- затребованные процессор ресурсы могут оставаться неиспользуемыми значительное время
- процессу не может быть известно заранее, какие ресурсы ему потребуются

Отсутствие перераспределения: 1) если процесс удерживает некоторые ресурсы и ему отказано в очередном запросе, то он должен освободить захваченные ресурсы и при необходимости запросить их все вновь. 2) если процесс затребовал ресурс, в

настоящий момент захваченный другом процессом, то ОС может вытеснить этот процесс и потребовать от него освободить захваченные им ресурсы.

Минусы:

- применимо только для ресурсов, которые можно легко сохранить, а позже восстановить, такие как процессор.

Циклическое ожидание: Можно избежать путем упорядочивания типов ресурсов. То есть если процесс захватил ресурс типа R, далее он может запросить только ресурсы, следующие согласно упорядочению за R.

Минусы:

- как и с удержанием и ожиданием, может быть неэффективной.

Устранение:

Подходы:

- не запускать процесс, если его запросы могут привести к взаимоблокировке
- не удовлетворять запросы процесса, если их выполнение способно привести к взаимоблокировке

Запрет: Рассмотрим систему из n процессов и m различных типов ресурсов.

Определим:

Ресурс: $\mathbf{R} = (R_1, R_2, \dots, R_m)$	Общее количество каждого ресурса в системе
Доступность: $\mathbf{V} = (V_1, V_2, \dots, V_m)$	Общее количество каждого ресурса, не выделенного процессам
Требования: $\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$	C_{ij} — запрос процессом i ресурса j
Распределение: $\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$	A_{ij} — текущее распределение процессу i ресурса j

С - матрица требований.

Должны выполняться следующие соотношения.

1. $R_j = V_j + \sum_{i=1}^n A_{ij}$ для всех j : все ресурсы либо свободны, либо выделены.
2. $C_{ij} \leq R_j$ для всех i и j : ни один процесс не может потребовать ресурса, превышающий его общее количество в системе.
3. $A_{ij} \leq C_{ij}$ для всех i и j : ни один процесс не может получить больше ресурсов, чем было им затребовано.

Стратегия:

Новый процесс $P_{\{n+1\}}$ запускается, только если

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^n C_{ij} \text{ для всех } j.$$

Это означает, что запуск нового процесса произойдет только тогда, когда могут быть удовлетворены максимальны требования всех текущих процессов + требования запускаемого процесса.

Запрет выделения ресурса (алгоритм банкира):

Безопасное состояние - состояние, в котором имеется по крайней мере одна последовательность, которая не приводит к взаимоблокировке. Состояние, не являющееся безопасным, называется **опасным состоянием**.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Матрица требований C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Матрица распределения A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	0	1	1

Вектор доступности V

a) Исходное состояние

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Матрица требований C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Матрица распределения A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	6	2	3

Вектор доступности V

б) Процесс P2 выполнен до завершения

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Матрица требований С

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Матрица распределения А

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	7	2	3

Вектор доступности V

б) Процесс P1 выполнен до завершения

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Матрица требований С

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Матрица распределения А

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	9	3	4

Вектор доступности V

г) Процесс P3 выполнен до завершения

Описанную концепцию иллюстрирует приведенный далее пример. На рис. 6.7, *a* показано состояние системы, образованной четырьмя процессами и тремя ресурсами. Общее количество ресурсов R1, R2 и R3 составляет соответственно 9, 3 и 6 единиц. В результате сделанного к этому моменту распределения ресурсов по процессам доступными остались по одной единице ресурсов R2 и R3. Безопасно ли данное состояние? Для ответа на этот вопрос зададимся другим вопросом, а именно — может ли какой-нибудь из четырех процессов быть выполнен при данных доступных ресурсах до завершения? Или, говоря иначе, могут ли при данном распределении ресурсов быть удовлетворены максимальные требования какого-то из процессов за счет оставшихся ресурсов? В терминах введенных ранее матриц и векторов для процесса *i* должно выполняться условие

$$C_{ij} - A_{ij} \leq V_j \text{ для всех } j.$$

Ясно, что для процесса P1 это невозможно, так как у него имеется только одна единица ресурса R1 и для полного удовлетворения ему требуется еще по две единицы ресурсов R1, R2 и R3. Однако если выделить процессу P2 одну единицу ресурса R3, то будут удовлетворены максимальные требования этого процесса и он сможет быть завершен. Когда процесс P2 оказывается завершенным, распределенные ему ресурсы могут быть возвращены в пул доступных. Это состояние системы показано на рис. 6.7, *b*. Вернувшись к вопросу о том, какой из процессов может быть завершен в данной ситуации, мы находим, что могут быть удовлетворены максимальные требования как процесса P1, так и процесса P3. Предположим, что мы выбрали процесс P1. По его завершении и возвращении захваченных им ресурсов в пул доступных ситуация будет выглядеть так, как показано на рис. 6.7, *c*. И наконец, по завершении процесса P3 мы придем к состоянию, изображенному на рис. 6.7, *d*. Наконец, мы можем завершить процесс P4. В этот момент все процессы завершены, и следовательно, исходное состояние (рис. 6.7, *a*) является безопасным.

Описанная концепция автоматически приводит к стратегии устранения взаимоблокировок, которая гарантирует, что система процессов и ресурсов всегда находится в безопасном состоянии. Когда процесс делает запрос к некоторому множеству ресурсов, предполагается, что запрос удовлетворен, после чего определяется, является ли обновленное состояние системы безопасным. Если это так, то запрос удовлетворяется; в противном случае процесс блокируется до тех пор, пока удовлетворение его запроса не станет безопасным.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Матрица требований С

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Матрица распределения А

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

С – А

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	1	1	2

Вектор доступности V

а) Исходное состояние

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Матрица требований С

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Матрица распределения А

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

С – А

	R1	R2	R3
	9	3	6

Вектор ресурсов R

	R1	R2	R3
	0	1	1

Вектор доступности V

б) Процесс P1 запрашивает по одной единице ресурсов R1 и R3

Рассмотрим состояние, определяемое матрицей на рис. 6.8, а. Предположим, что процесс P2 делает запрос на одну дополнительную единицу ресурса R1 и одну — ресурса R3. Если предположить, что запрос будет выполнен, то в результате состояние системы станет таким, как показано на рис. 6.7, а. Мы уже видели, что это состояние является безопасным. Таким образом, запрос можно удовлетворить. Теперь вернемся к состоянию, представленному на рис. 6.8, а, и предположим, что такой же запрос на одну дополнительную единицу ресурса R1 и одну — ресурса R3 делает процесс P1. Если этот запрос будет удовлетворен, состояние системы станет таким, как показано на рис. 6.8, б. Является ли это состояние безопасным? В данном случае ответ — “нет”, поскольку каждому из процессов требуется по одной дополнительной единице ресурса R1, а в наличии свободных единиц этого ресурса уже нет. Следовательно, запрос процесса P1 должен быть отклонен, а сам процесс — блокирован.

Важно отметить, что состояние, представленное на рис. 6.8, б, не является взаимоблокировкой. Оно всего лишь может привести к ней. Например, при работе процесс P1 может освободить по одной единице ресурсов R1 и R3, и система вновь вернется в безопасное состояние. Следовательно, стратегия устранения взаимоблокировок не занимается точным предсказанием взаимоблокировок; она лишь предвидит их возможность и устраняет ее.

При использованы не нужны ни перераспределение, ни откат процессов, как в случае обнаружения. Кроме того накладываются меньше ограничений.

Обнаружение:

Алгоритмы:

- При каждом запросе. Плюсы - Раннее обнаружение и упрощение алгоритма.
Минусы - заметное потребление процессорного времени.
- Менее часто: по специальному алгоритму.

Для алгоритма используется матрица распределения и вектор доступности.

Кроме того, определена матрица запросов \mathbf{Q} , такая, что Q_{ij} представляет собой количество ресурсов типа j , затребованное процессом i . Алгоритм работает, помечая незаблокированные процессы. Изначально все процессы не помечены. После этого выполняются следующие шаги.

1. Помечаем все процессы, строки в матрице распределения которых состоят из одних нулей.
2. Временный вектор \mathbf{W} инициализируем значениями вектора доступности.
3. Находим индекс i , такой, что процесс i в настоящий момент не помечен и i -я строка матрицы \mathbf{Q} не превышает \mathbf{W} , т.е. для всех $1 \leq k \leq m$ выполняется $Q_{ik} \leq W_k$. Если такой строки нет, алгоритм прекращает свою работу.
4. Если такая строка имеется, помечаем процесс i и добавляем соответствующую строку матрицы распределения к \mathbf{W} , т.е. выполняем присвоение $W_k = W_k + A_{ik}$ для всех $1 \leq k \leq m$. Возвращаемся к шагу 3.

Взаимоблокировка имеется тогда и только тогда, когда после выполнения алгоритма есть непомеченные процессы. Множество непомеченных процессов в точности соот-

ветствует множеству заблокированных процессов. Стратегия этого алгоритма состоит в поиске процесса, запросы которого могут быть удовлетворены доступными ресурсами, а затем предполагается, что эти ресурсы ему выделены и процесс, завершив свою работу, освобождает их. После этого алгоритм приступает к поиску другого процесса, который может успешно завершить свою работу. Заметим, что данный алгоритм не гарантирует предотвращения взаимоблокировок — это зависит от порядка удовлетворения запросов процессов. Все, что делает данный алгоритм, — это определяет, имеется ли взаимоблокировка в настоящий момент.

Для иллюстрации алгоритма обнаружения взаимоблокировок рассмотрим рис. 6.10. Алгоритм работает следующим образом.

1. Помечаем P4, поскольку этот процесс не имеет распределенных ему ресурсов.
2. Устанавливаем $\mathbf{W} = (0 \ 0 \ 0 \ 0 \ 1)$.
3. Запрос процесса P3 не превышает \mathbf{W} , так что помечаем P3 и устанавливаем $\mathbf{W} = \mathbf{W} + (0 \ 0 \ 0 \ 1 \ 0) = (0 \ 0 \ 0 \ 1 \ 1)$.
4. Других непомеченных процессов, строки матрицы \mathbf{Q} которых не превышают \mathbf{W} , нет. Алгоритм прекращает свою работу.

Таким образом, алгоритм позволяет заключить, что процессы P1 и P2 взаимно заблокированы.

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Матрица запросов Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Матрица распределения A

	R1	R2	R3	R4	R5
	2	1	1	2	1

Вектор ресурсов

	R1	R2	R3	R4	R5
	0	0	0	0	1

Вектор доступности

Восстановление (стратегии):

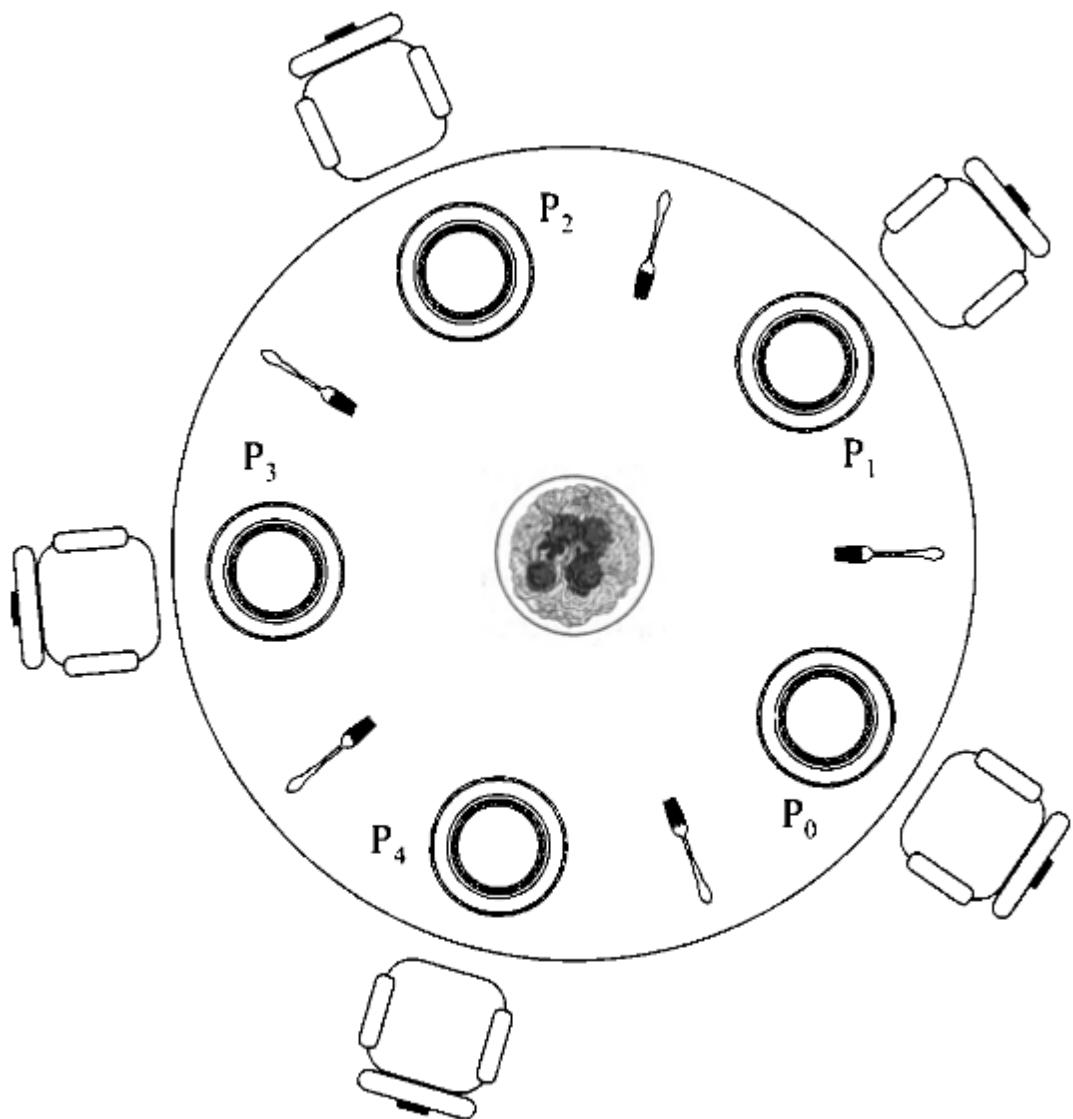
- Прекратить выполнение всех заблокированных процессов (самая распространенная в оси).
- Вернуть каждый из заблокированных процессов в некоторую ранее определенную точку и перезапустить все процессы (должны быть встроены механизмы).
- Последовательно прекращать заблокированные процессы по одному. Порядок базируется на каком-то критерии.
- Последовательно перераспределять ресурсы до тех пор, пока взаимоблокировка не прекратится. Также нужен критерий и реализована возможность отката к состоянию, в котором он находился до получения ресурса.

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 подточные 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.



Решение с использованием семафоров:

```

/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher(int i)
{
    while (true)
    {
        think();
        wait(room);
        wait(fork[i]);
        wait(fork [(i + 1) mod 5]);
        eat();
        signal(fork [(i + 1) mod 5]);
        signal(fork[i]);
        signal(room);
    }
}
void main()
{
    parbegin(phiosopher(0), philosopher(1),
             philosopher(2), philosopher(3),
             philosopher(4));
}

```

На рис. 6.12 предложено решение этой задачи с использованием семафоров. Каждый философ, садясь за стол, сначала берет левую вилку, а затем правую. После того как философ пообедает, использованные им вилки заменяются. Увы, такое решение может привести к взаимоблокировке, если философы, одновременно проголодавшись, все вместе сядут за стол и одновременно возьмут лежащие слева вилки. В этой неприятной ситуации им придется голодать.

Чтобы избежать риска взаимоблокировки, можно купить еще пять вилок (кстати, самое подходящее решение задачи с точки зрения гигиены!) или научить философов есть спагетти одной вилкой. Еще один подход состоит в том, чтобы нанять вышибалу, который не позволит пяти философам садиться за стол одновременно. Если же за столом соберутся не более четырех философов, то по крайней мере один из них сможет воспользоваться двумя вилками. На рис. 6.13 приведено соответствующее решение задачи (вновь с использованием семафоров). Ни взаимоблокировок, ни голодания при таком решении просто не может быть.

Решение с использованием монитора:

```

monitor dining_controller;
cond ForkReady[5];           /* Условная переменная для синхронизации */
boolean fork[5] = {true};    /* Состояние доступности каждой вилки */
void get_forks(int pid)      /* pid - идентификатор философа */
{
    int left = pid;
    int right = (++pid) % 5;

    /* Предоставление левой вилки */
    if (!fork[left])
        cwait(ForkReady[left]);      /* Очередь условной переменной */

    fork[left] = false;

    /* Предоставление правой вилки */
    if (!fork[right])
        cwait(ForkReady[right]);      /* Очередь условной переменной */

    fork[right] = false;
}

void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;

    /* Освобождение левой вилки */
    if (empty(ForkReady[left]))      /* Этую вилку никто не ждет */
        fork[left] = true;
    else /* Пробуждение процесса, ожидающего эту вилку */
        csignal(ForkReady[left]);

    /* Освобождение правой вилки */
    if (empty(ForkReady[right]))      /* Этую вилку никто не ждет */
        fork[right] = true;
    else /* Пробуждение процесса, ожидающего эту вилку */
        csignal(ForkReady[right]);
}

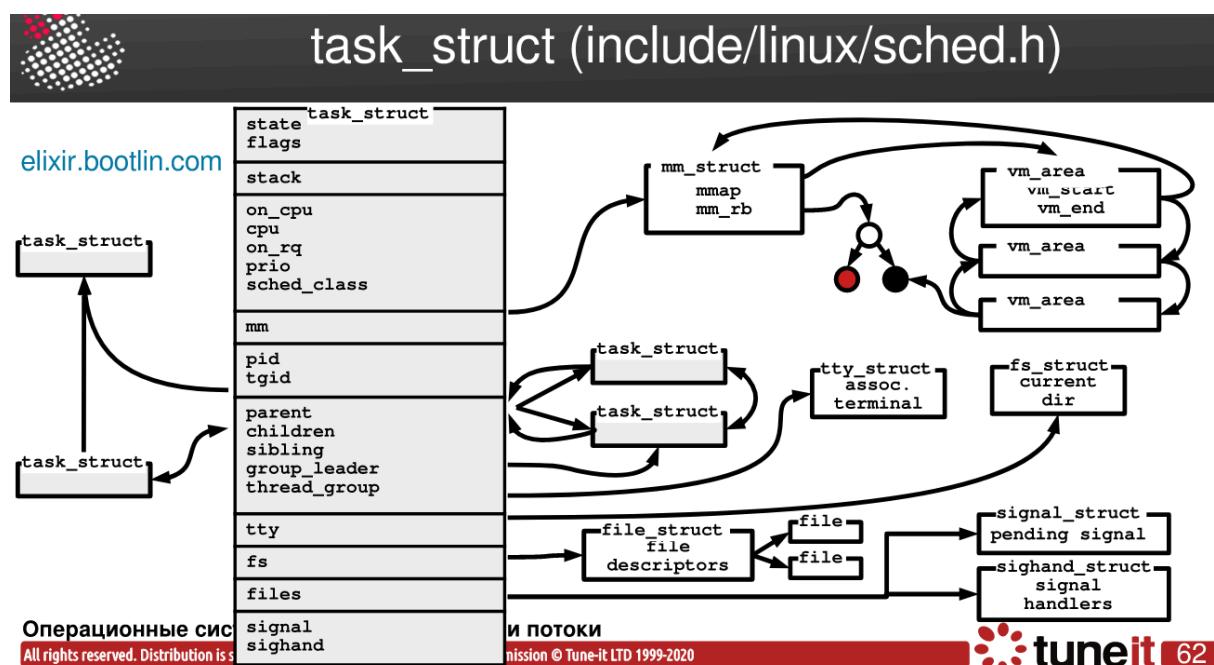
void philosopher[k = 0 to 4]      /* Пять философов-клиентов */
{
    while (true)
    {
        <Размышления>;
        get_forks(k);            /* Запрос вилок с помощью монитора */
        <Поедание спагетти>;
        release_forks(k);        /* Освобождение вилок с помощью монитора */
    }
}

```

тора. В программе определен вектор из пяти условных переменных, по одной условной переменной на вилку. Эти условные переменные используются для того, чтобы обеспечить ожидание философом доступности вилки.

Кроме того, имеется вектор логических значений, который записывает состояние доступности каждой вилки (`true` означает, что вилка доступна). Монитор состоит из двух процедур. Процедура `get_forks` используется философом для захвата его левой и правой вилок. Если любая из вилок недоступна, процесс философа помещается в очередь соответствующей условной переменной. Это позволяет другим процессам философов войти в монитор. Процедура `release_forks` используется для освобождения двух вилок. Обратите внимание, что структура этого решения похожа на структуру решения с семафорами, предложенного на рис. 6.12. В обоих случаях философ сначала завладевает левой вилкой, а затем — правой. В отличие от решения с использованием семафора рассматриваемое решение с использованием монитора не страдает от взаимоблокировки, потому что в мониторе одновременно может находиться только один процесс. Например, первый вошедший в монитор процесс философа гарантированно сможет забрать вилку справа (после того, как выберет левую вилку) до того, как следующий философ справа получит шанс взять свою левую вилку (являющуюся правой вилкой для первого философа).

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



В структуре `task_struct` присутствуют следующие поля.

- **state** - текущее состояние процесса.
- **flags** - “флаги” - необходимы планировщику и другим различным системам, которые в зависимости от значений флагов осуществляют выбор.
- **stack** - ссылка указателя на пользовательский стек рассматриваемого процесса.
- **on_cpu** описывает момент нахождения процесса на процессоре;
- **on_rq** (runnable queue) означает нахождение в состоянии ожидания процесса.
- **prio, sched_class** - приоритет, а также класс, обеспечивающий диспетчеризацию.

- **mm** - структуры, описывающие адресное пространство (АП) процесса
- **pid** - это идентификационный номер процесса, представляющий собой число
- **tgid** - номер “Thread Group ID”. В Linux нет тредов в качестве структур ядра, и все треды, связанные с одним процессом, будут в Linux иметь один и TID.
- **parent** - это процесс, породивший именно наш процесс.
- Ниже описываются ссылки на детей каждого из потоков (**children**) и некоторые другие родственные связи - **sibling** (братья/сестры).
- **group_leader** - первый поток, запущенный в процессе.
- **thread_group** - ссылки на все потоки, которые были порождены внутри нашего процесса.
- **tty** - ссылка на терминал (ввод, вывод и ошибка для процесса). У демонов отсутствует.
- **fs** - ссылка на файловую систему, где находится текущая директория.
- **files** - файлы, дескрипторы.
- **signal, sighand** - переменные для сигналов и обработчиков сигналов.

mm_struct - общая структура, описывающая АП; организованы в двухсвязный список

- **mmap** - указывает на **vm_area**
- **mm_rb** - красно-чёрное дерево **vm_area** (указывает на голову)

vm_area - области виртуального АП (сегменты проги)

- **vm_start, vm_end** - начало и конец области
- Организованы в виде двусвязного списка внутри ядра. И при этом существует “красно-чёрное дерево”, которое позволяет ускорить поиск внутри этих структур.

И так далее, там много неосвещенных полей.

85. Диаграмма состояния процесса Linux.



На иллюстрации “большими” буквами приведены состояния, описанные на предыдущем слайде в разделе state. В Linux, кроме **TASK_RUNNING**, для описания , где находится процесс, ничего больше нет, поэтому в дальнейшем мы не знаем, как рассматриваемый процесс в Linux продолжает свою работу, и что с ним конкретно происходит.

Состояния **TASK_INTERRUPTIBLE** и **TASK_UNINTERRUPTIBLE** обычно используются во время различных видов ожидания: IO, блокировок и т.д. Первому могут прийти прерывания, второму нет.

TASK_RUNNING:

- **on_rq** - в очереди ожидания
 - **on_cpu** - на процессоре

Процесс сам себя диспетчеризует. Каждые 4-6 мс происходит прерывание по таймеру, что приводит к вызову функции **scheduler_tick()**. Далее осуществляется проверка: если в очереди ожидания есть более высокоприоритетные процессы, то процесс сам вызывает функцию **put_prev_task()**, после чего сам помещает себя в очередь ожидания, переключает контекст и освобождает процессор для другого процесса.

Обратное действие осуществляется функцией **schedule()**. Т.е. если ядро решает, что необходимо пробудить процесс, находящийся в `rq`, то это происходит с помощью **schedule()**, которая вызывает установку на CPU следующей задачи. После этого происходит вызов функции **context_switch()**, которая переключает регистры. И далее на CPU начинает находится поток, который будет исполняться до следующего тика, после чего опять произойдет процесс принятия решения: будет ли процесс исполняться на CPU дальше, или произойдёт очередное переключение на следующий в очереди процесс.

При создании процесса ему задается состояние **TASK_NEW**. После того, как процесс становится готов к исполнению, вызывается функция **wake_up_new_task()**. Затем процесс попадает в **RQ**

ZOMBIE - завершился, но находится в списке, т.к. **parent** ещё не считал код выполнения.

TASK_DEAD говорит, что структуру уже можно переиспользовать. Завершился полностью.

deactivate_task() - перевод в ожидание.

do_exit, release_task() - перевод в **EXIT_ZOMBIE** или **TASK_DEAD**.

86. Создание процесса Linux на уровне пользовательского процесса.



Userland: Создание процесса

- **fork()** - создать процесс
 - Возвращает номер процесса родителю, 0 — созданному процессу, -1 в случае ошибки.
- **vfork()** - создать процесс с копией адресного пространства родителя
 - Родительский процесс блокирован до тех пор, пока дочерний не вызовет `exit()` или `execve()`
- **clone()** - создать процесс, управляя копированием выбранных частей процесса
 - `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);`
- **execv(const char *path, const char *arg, ...)** - перекрытие образа процесса

fork() - системный вызов для создания/ветвления процесса, используемый в Unix(-like) ОС. Создает копию процесса, возвращает родителю ее номер, а ей 0 или -1 в случае ошибки (ее код сохранится в **errno**). Жирная, в Linux слегка бесполезная, т.к. всё равно вызывает **clone()** с параметрами, как и **create_thread()**.

В **fork()** создаются лишь наиболее критически важные страницы АП, а все остальные только при необходимости (когда их нужно будет загрузить в ядро). Сегменты из род. процесса копируются лишь в виде записей о них ("в реальности" они будут создаваться только по обращении или изменении данных - COW).

Отдельный скрипт не создавали, потому что часто приходилось дублировать код и данные, и в таком случае можно было ничего не делать вручную (например, когда в сервере создаётся процесс под каждый запрос).

Unix - **vfork()**, ускоряет порождение процесса за счёт использования им АП родительского, который блокируется до **exit()/execve()** дочернего.

Linux - **clone()**, ускоряет порождение процесса за счёт управления копированием выбранных заданием опред. флагов частей процесса. Используются для потоков (на уровне ядра явл. **task_struct**).

execv() - запуск программы по выбранному пути с аргументами с заменой образа процесса.

87. Создание и завершение процесса Linux на уровне ядра.

Вызываемые функции.



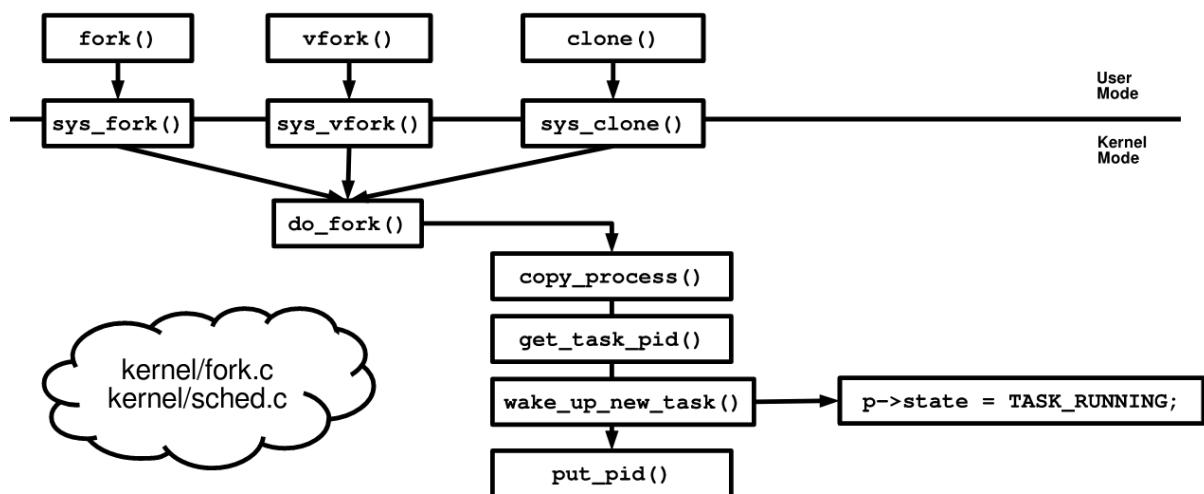
- В ядре Linux нет примитива Thread, соответствующего пользовательскому потоку
 - Соответственно нет планирования потоков
- Потоки создаются так же, как и процессы
 - Разделяют ресурсы вызвавшего fork процесса
 - Каждый поток — отдельная task_struct
 - Создание: `clone(CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND | CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM, 0); (NPTL version)`

В Linux нет возможности создать тред, однако создать создать так называемое подобие треда можно с помощью системного вызова `clone()`.

NPTL - native posix thread library.

В Linux есть понятие “тредовой группы”. Туда относятся все pid's, которые были созданы как треды внутри рассматриваемого процесса. У всех тредов, созданных внутри линуксового ядра, имеется общая группа потоков, но они работают абсолютно независимо. Для диспетчеризации также используется `task_struct`.

Создание процесса



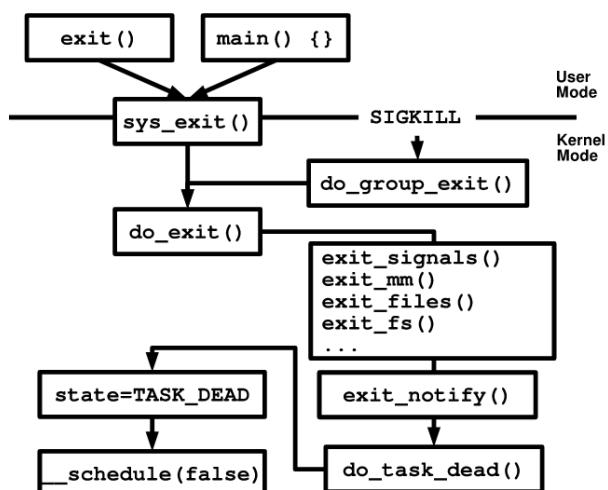
Все системные вызовы вызывают `do_fork()`. Она:

- `copy_process()` - копирует процесс в зависимости от существующих опций.
- `get_task_pid()` - получает для процесса новый идентификатор
- `wake_up_new_task()` - переводит процесс в состояние `TASK_RUNNING`.
- `put_pid()` - укладываем pid в структуры, которые будут управлять процессом.

UserMode - уровень пользователя, KernelMode - уровень ядра.

// `copy_process()` - копирование процесса (опущено за ненадобностью)

Завершение процесса (kernel/exit.c)



- Нормальное завершение
 - окончание `main()`
- Если родитель завершился раньше дочернего (`zombie`) нужно найти кому отдать код
 - Процессу в группе
 - `init`

Основной способ завершения процесса: `exit(code)`, по code определяется код возврата или завершение функции `main`.

`do_group_exit()` - вся threadовая группа, у которой владелец текущий, должна быть завершена. Короче убиваем всех детей.

Основная причина процесса “зомби” в том, что процессу, который завершается, некому отдать код возврата.

После do_exit() удаляются все структуры. exit_notify() - оповестить предка. do_task_dead() - перевод состояния процесса в task_dead(). Далее управление передается в планировщик, который забывает об этом процессе.

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



- Предназначен для выполнения фоновых операций в ядре
 - Планировщик работает с ними так же, как с процессами
- Нет своего адресного пространства
 - mm в task_struct равен NULL
- kthread являются потомками kthreadd (pid==2)
 - Все фоновые потоки ядра ps --ppid=2
- Создание — kthread_create()
 - После создания нужно выполнить wake_up_process()
 - Одновременное создание и запуск макрос kthread_run()
- Для остановки другие потоки вызывают kthread_stop()
 - А сам поток должен проверять kthread_should_stop()

kthread (kernel thread) - процесс специального назначения, для выполнения фоновых операций. Нет адресного пространства, так как поскольку это kernel process, то у него адресное пространство - ядро.

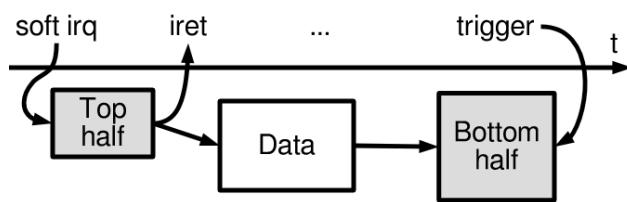
В отличие от process exec(), когда процесс автоматически переходит в состояние RUNNABLE, для запуска kthread необходимо вызвать функцию wake_up_process().

Нюанс: другие потоки не могут остановить kthread. Если другие процессы хотят остановить его, то они вызывают kthread_stop(), однако при этом сам kthread не останавливается, а лишь получает пометку, что его хотят остановить. Внутри самого потока ядра эта пометка должна проверяться функцией kthread_should_stop(), и если функция возвращает значение, говорящее, что процесс надо остановить, то необходимо самостоятельно завершить поток.

kthread_create() может вызвать только на уровне ядра. Надо для всяких драйверов и модулей.

Tasklets

```
include/linux/interrupt.h
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```



- Обслуживание «bottom half» для обработки программного прерывания (soft irq)
- Один tasklet одновременно на одном CPU, разные одновременно на разных CPU
- Два приоритета — normal и hi
 - tasklet_schedule и tasklet_hi_schedule
 - «Hi» выполняются раньше
- count - 1 - деактивирован

Предназначение - разнести обработку программного прерывания soft irq. Другими словами задача - обработать задачку отложено в асинхронном режиме.

Снизу картинки: надо выполнить программное прерывание.

- top half - регистрация прерывания как можно быстрее (взять структуру и сделать возврат)
- bottom half - вызывается по какому-то триggerу (по времени, после обработки таймера прерывания в ядре ...). Таким образом сама обработка отсрочивается и позднее обрабатывается в асинхронном виде по отношению к самому запросу прерывания.

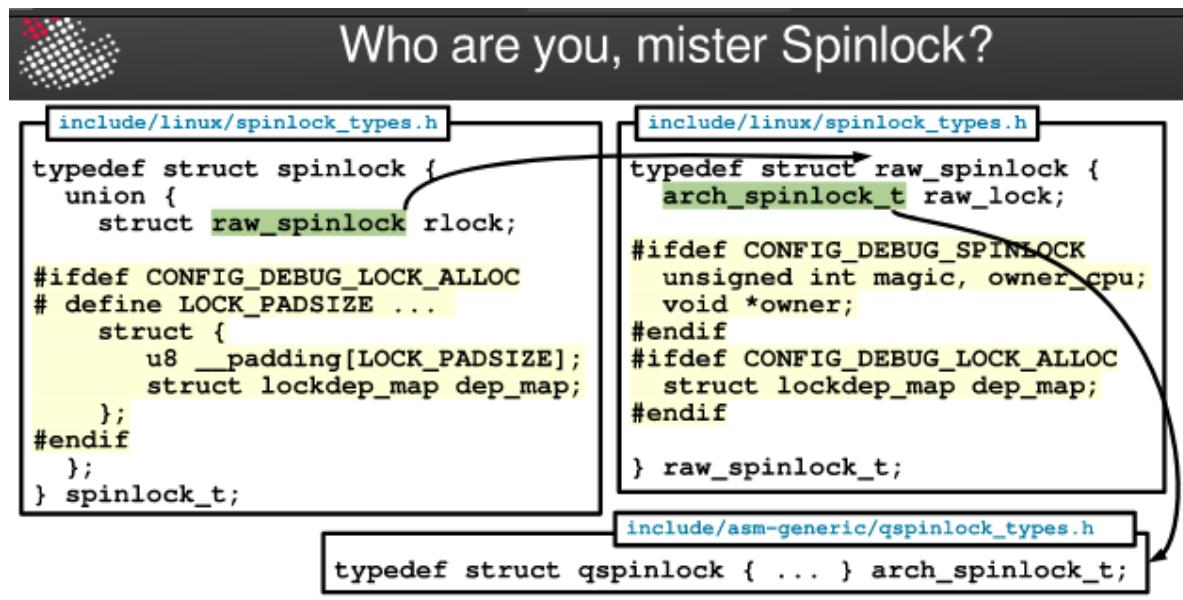
Структура tasklet (обычно привязываются к конкретному процессору и не могут существовать на разных цпу, то есть у каждого цпу есть своя очередь tasklet):

- func - сама функция, которую хотим исполнить
- data - какие-либо данные
- count - предназначена для активации/деактивации tasklet.

Приоритеты задаются разными вызовами: tasklet_schedule и tasklet_hi_schedule.

- tasklet_schedule - системный вызов, который позволяет запланировать вызов в определенный момент времени или по указанному событию
- tasklet_hi_schedule - обрабатывает более высокоприоритетные tasklet.
- Те, которые hi будут обрабатываться раньше остальных

89. Примитивы синхронизации Linux. Spinlock и qspinlock.



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

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.

Описание слайда:

На рисунке выше представлена структура `spinlock`. Она состоит из `union`'а, первым полем которого является `raw_spinlock` или, если включен режим `CONFIG_DEBUG_LOCK_ALLOC`, то она раскрывается в то, что между дефайнами.

Справа представлена структура `raw_spinlock`. В случае если включен дефайн `CONFIG_DEBUG_SPINLOCK`, то в структуру в качестве отладочной информации добавляется `magic` (вроде бы идентификатор), `owner_cpu` - на каком процессоре исполняется. Также если включен режим `CONFIG_DEBUG_LOCK_ALLOC`, то в структуру дополнительно включается карта, которая устанавливает "взаимозависимости спинлоков друг от друга". Это необходимо, когда наступает дедлок, чтобы разобраться, в чем заключается причина наступившего события.

Также в структуру raw_spinlock включается arch_spinlock_t, который зависит от используемой архитектуры. Каждая архитектура предпочитает использование тех типов физической реализации спинлоков, которые наиболее подходят для данной конкретной платформы. Так для архитектуры x86 arch_spinlock_t раскрывается в qspinlock (спинлок с очередью).



Реализация и операции

```
include/asm-generic/qspinlock_types.h
typedef struct qspinlock {
    union {
        atomic_t val;
    #ifdef __LITTLE_ENDIAN
        struct {
            u8 locked;
            u8 pending;
        };
        struct {
            u16 locked_pending;
            u16 tail;
        };
    #else
        // reverse order and alignment
    #endif
    };
} arch_spinlock_t;
```

- spin_lock_init
- spin_lock
- spin_lock_bh
 - disables software interrupts and lock
- spin_lock_irqsave
 - disables irq for local CPU and save state
- spin_lock_irq
- spin_unlock
- spin_unlock_bh
- spin_is_locked
- ...

val - атомарное значение, использующееся для определения, занят ли спинлок.

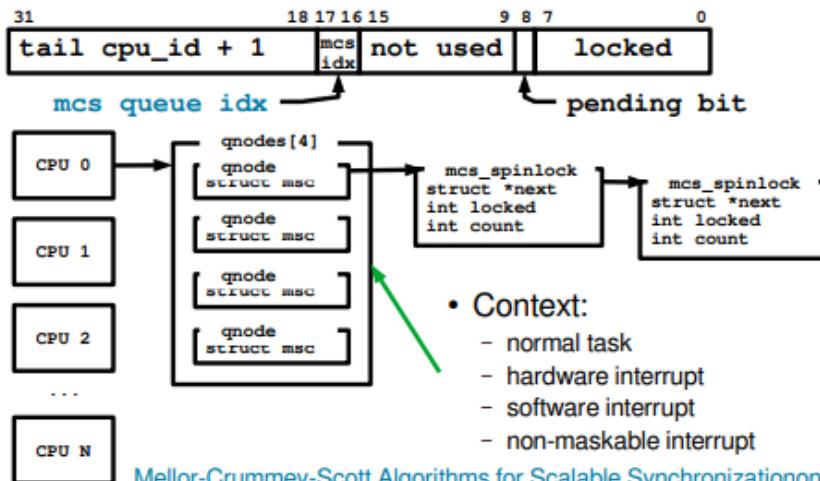
В зависимости от конфигурации (LITTLE_ENDIAN, BIG_ENDIAN) в структуре будет различный порядок полей.

Операции:

- spin_lock_init - инициализация спинлока
- spin_lock - блокировка
- spin_lock_bh (bottom_half) - блокировка с запретом программных прерываний (бillet 88)
- spin_lock_irq - с запретом аппаратных прерываний без сохранения state прерывания
- spin_lock_irqsave - с запретом с сохранением
- И т.д.



Qspinlock



Mellor-Crummey-Scott Algorithms for Scalable Synchronization on Shared Memory Multiprocessors
Операционные системы. Часть 2. Процессы и потоки
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



mcs - mellor crummey scott (автор)

Рассмотрим структуру 32-битного атомика:

- locked - указывает, заблокирован ли ресурс или нет. Если свободен, то 0, иначе 1.
- pending_bit - говорит о том, что спинлок уже захвачен, и уведомляет, что вот не становись в очередь, так как это довольно затратная операция, а подожди, сейчас скоро освобожусь.
- mcs_idx - выбирает одну из 4 очередей (контекстов с точки зрения состояния обработки прерываний). В зависимости от текущего уровня "привилегий" ядра, задача добавляется в определенную очередь
 - normal task - для обычных задач, которые могут быть прерваны аппаратными или программными прерываниями
 - hardware - в аппаратном прерывании
 - software - в программном
 - non-maskable - высокоприоритетное прерывание
- tail - старшая часть, определяет ссылку на очередь, что позволяет сделать процесс ожидания более эффективным, сокращая время доступа к памяти.

В зависимости от текущего состояния пути, выбирается тот или иной путь прохождения обработки.

- Fast path (uncontented) - спин лок свободен, сразу занимаем.
- Slow path:
 - Pending - спин лок занят, но никто не стоит в ожидании (`pending_bit = 1`)
 - Uncontented queue - очередь содержит небольшое количество элементов
 - Contented queue - в очереди большое количество элементов.

Сама процедура кручения осуществляется на локе, который будет находиться в памяти, загруженной в кеше каждого процессора: для этого та переменная, на которой осуществляется ожидание, выносится в кеш, чтобы сократить доступ к shared memory.

90. Примитивы синхронизации Linux. Semaphore и Mutex.



Semaphore

```
include/linux/semaphore.h

struct semaphore {
    raw_spinlock_t lock;
    unsigned int count;
    struct list_head wait_list;
};

#define __SEMAPHORE_INITIALIZER(name, n) \
{ \
    .lock    = __RAW_SPIN_LOCK_UNLOCKED((name).lock), \
    .count   = n, \
    .wait_list = LIST_HEAD_INIT((name).wait_list), \
}
static inline void sema_init(struct semaphore *sem, int val)
{
    static struct lock_class_key __key;
    *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
    lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
}
```

- void **down**(struct semaphore *sem);
- void **up**(struct semaphore *sem);
- int **down_interruptible**(struct semaphore *sem);
- int **down_killable**(struct semaphore *sem);
- int **down_trylock**(struct semaphore *sem);
- int **down_timeout**(struct semaphore *sem, long jiffies);

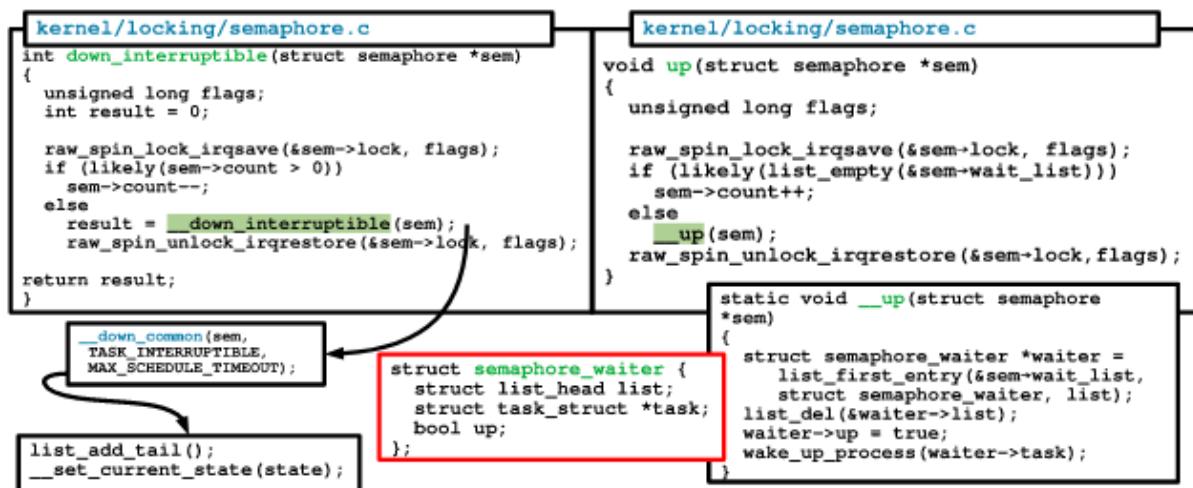
- `raw_spinlock_t lock` - защита от изменения
- `count` - сколько есть ресурса
- `wait_list` - сюда будет подставлен заголовок связанного списка, на котором будут ожидать трэды, которые пришли в тот момент, когда семафор испытывает contention (борьбу за ресурсы)

Функции:

- `down, up` - занять, освободить ресурс
- `down_trylock` - проверяем, свободен или нет. Если свободен - занимаем
- `down_killable` - если убить программу при помощи `kill`, то семафор должен освободиться
- `down_timeout` - ожидать на семафоре ограниченное количество времени. `jiffies` - тики таймера. Когда тики истекут, то выйдем из состояния ожидания.

Здесь нет “агрессивной оптимизации кода”

Semaphores up/down



`down_interruptible`: Если количество ресурсов больше нуля, то семафор декрементируется. В противном случае происходит вызов процедуры, помещающей поток в состояние ожидания.

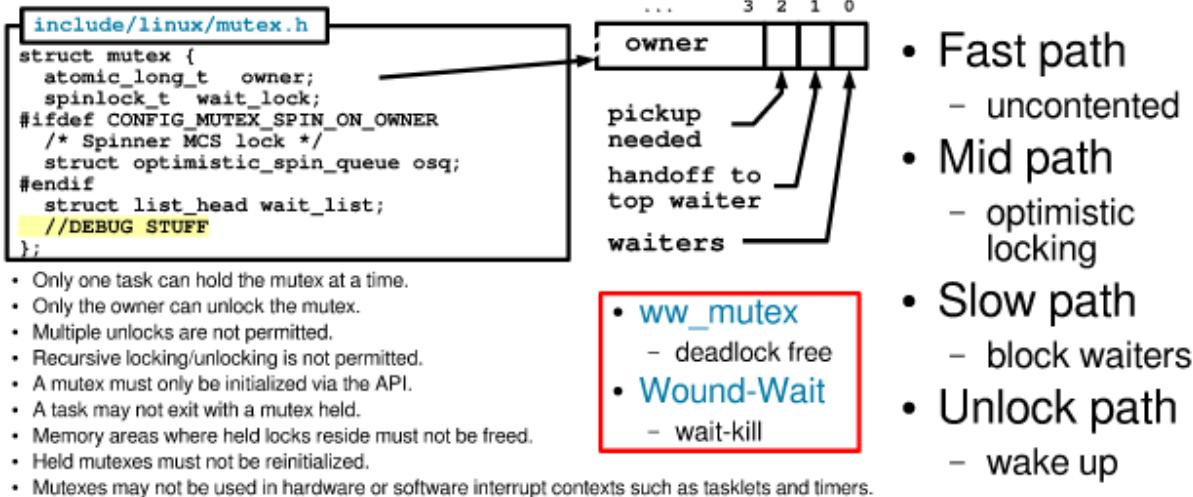
`semaphore_waiter` - структура, в которую помещаются задачи, ожидающие освобождения ресурсов.

`up`: если очередь пустая, то увеличиваем счетчик. В противном случае достаем с головы очереди задачу и пробуждаем ее.

`likely` - макрос компилятора, который говорит, что текущий бранч будет вероятным.

`raw_spin_lock_irqsave` и т.п. - с запретом аппаратных прерываний с сохранением `state`.

Mutex



Структура:

- `owner` - адрес в памяти, который вызвал блокировку спинлока.

- pickup needed, handoff to top waiter - позволяют передавать мьютекс без дополнительных действий первому ожидающему его треду.
- waiters - есть кто-то, кто ожидает.
- Позволяет ускорить процесс обработки: мьютекс можно не освобождать, а заглянуть в очередь и передать первому нуждающемуся.
- wait_lock - защищает очередь optimistic_spin_queue
- osq - msc lock (билет 89)

Пути:

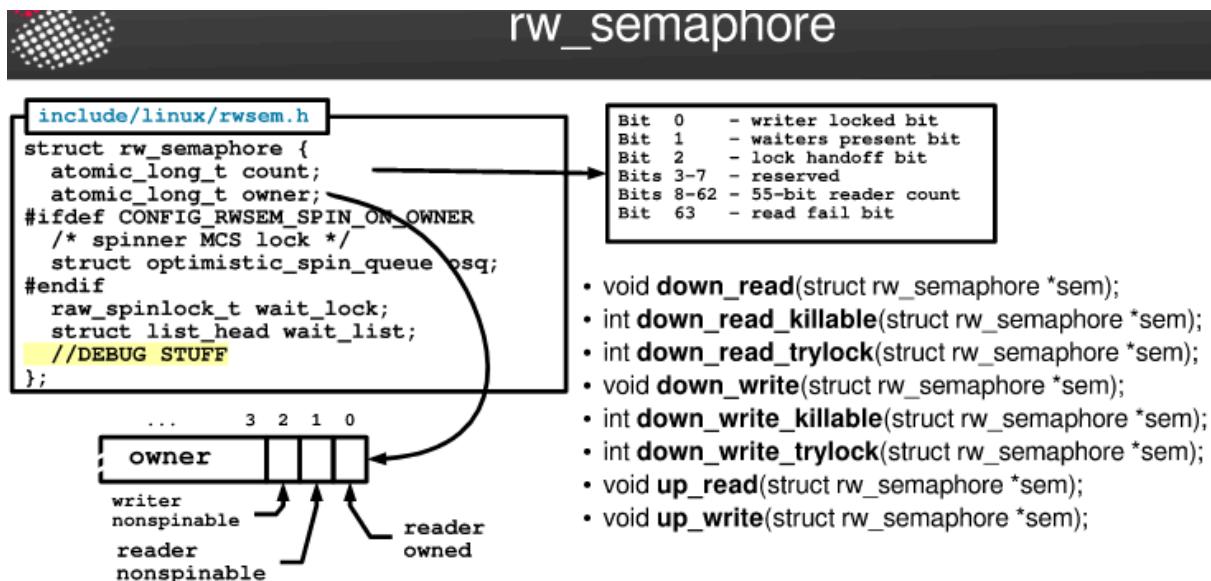
- Fast - uncontented - не занят.
- Mid - учитывает приоритеты в msc очередях каждого процессора, ожидая на спин блокировке.
- Slow - тот, кто ожидает очередь, уже блокирован, следовательно и мы блокируемся.
- Unlock path - разблокируемся: проверяем, где находятся ожидающие треды: тем, которые работают будет дан приоритет, а те, которые заблокированы (если нет высокоприоритетных тредов), будут разбужены.

Слева снизу: условия использования.

ww_mutex - deadlock free. Как в бд транзакции в зависимости от времени заставляем некоторую транзакцию переделать свою работу.

- wound-wait, wait-kill - алгоритмы.

91. Примитивы синхронизации Linux. rw_semaphore, seqlock.



Необходим, чтобы организовать сообщество “читателей и писателей”, чтобы большее количество читателей могло осуществлять чтение, а небольшое количество писателей могли осуществлять запись. Причем в каждый момент времени запись может осуществлять только 1 писатель.

Структура:

- count - описывает некоторое число дополнительных битов для организации читателей и писателей.
 - lock handoff bit - также используется для передачи блокировки (билет 90)
 - 55-bit reader count - 1 бит - 1 читатель
 - read fail bit - количество читателей переполнилось
- owner - объекты, захватив
 - reader owned - читатель или писатель
 - writer/reader nonspinable - показывает что reader, writer не могут спиниться на этой блокировке

Функции (те же самые как в билете выше (копия)):

- down, up - занять, освободить ресурс
- down_trylock - проверяем, свободен или нет. Если свободен - занимаем
- down_killable - если убить программу при помощи kill, то семафор должен освободиться



Sequence counters and seqlock

- Механизм для реализации неблокирующего чтения (с повторами) и записи без голодания
- Несколько версий
 - seqcount_t — запись должна быть синхронизирована внешними средствами
 - seqcount_LOCKTYPE_t — запись синхронизирована средствами выбранного LOCKTYPE
 - seqcount_t с семантикой защелки (две копии данных для четного/нечетного значения счетчика)
 - seqlock_t — запись синхронизирована spinlock и «non preemptible»

```
do {
    seq = read_seqbegin(&foo_seqlock);
    /* [[read-side critical section]] */
}
while (read_seqretry(&foo_seqlock, seq));
```

Sequence counters and sequential locks

Основное назначение - неблокирующее чтение, запись “без голодания”. В предыдущем примитиве может возникнуть ситуация, что writer будет очень долго ждать reader'ов, что не всегда хорошо.

В период чтения данных может произойти запись. Поэтому блокировка при помощи специального метода read_seqretry сообщит, что “читателю” нужно повторить операцию чтения. Цикл будет до тех пор, пока не станет свободной блокировка от операции записи. (представлено в рамке)

Версии:

- seqcount_t - нет блокировки на запись, и следует “ручками” прописать последовательность записи, а сама блокировка по умолчанию имеет блокировки только на чтение.
- seqcount_LOCKTYPE_t - вместо LOCKTYPE подставить тип переменной, которая будет использоваться для блокирования записи

- seqcount_t - в этой блокировке есть счетчик, который содержит две копии данных для четного и нечетного значения счетчика. (напр. четный - для чтения, нечетный - для записи).
- seqlock_t - синхронизирована спинлоком и запрещена для прерываний

92. Типы процессов и потоков Windows.

Типы процессов

- «Современные процессы»
 - Universal Windows Platform (UWP) processes (AKA Immersive processes) — Windows 8 (Windows Store)
 - Protected processes (Windows Media Certificate)
 - + Protected Processes Light
- Minimal processes
 - Нет пользовательского адресного пространства
 - System process и Memory Compression process
- Pico-процессы
 - Drawbridge project (Pico providers, например Lxss.sys и LxCore.sys)
 - Ограничивают доступ к системным функциям, предоставляют набор callback
- Trustlets - Isolated User Mode (IUM) Processes
 - используют виртуализацию VTL1 (Virtual Trust Level 1), VTL0 — остальная система
- Windows on Windows (WOW) - 32 бита в 64 битном режиме
- JOBS — средство группировки процессов
- Dos, Win16, bat-файлы

Современные процессы:

- UWP используют те приложения, которые ставятся нативно (через Windows Store)
- Protected процессы - работают с защищенным содержимым (чтобы нельзя было спиратить после декодирования). Они подписываются специальным сертификатом. Если есть права администратора, но вы не запросили полномочия внутри системы, то вам будет отказано в доступе к получению образа этих систем.

Minimal processes:

- Похожи на UNIX kthread.
- Нет пользовательского адресного пространства - работают как часть ядра.

Pico-процессы:

- Для взаимодействия с ядром ограниченный доступ к системным вызовам.
- Необходимо установка специальных пико-провайдеров.
- Подсистема Linux for Windows реализована через эти процессы.

Trustlet:

- Предназначение - запуск процессов в контейнере виртуализации.

Пул потоков — это коллекция рабочих потоков, которые эффективно выполняют асинхронные обратные вызовы от имени приложения. Пул потоков используется главным образом для сокращения числа потоков приложения и управления рабочими потоками.

Dos, Win16, bat-файлы, Posix - для обеспечения совместимости. Запускаются через отдельную подсистему

Windows on Windows (WOW) - поддержка 32-битной адресации на 64-битной машине. Тоже для совместимости.

Типы Потоков

- Обычные потоки (1:1 User-Kernel)
 - В т.ч. базовая реализация posix threads
- Fiber — чисто пользовательская реализация потоков, невидимы ядру
 - Kernel32.dll (ConvertThreadToFiber, CreateFiber)
 - Cooperative multitasking, совместно используют один контекст потока ядра
- User-mode scheduling threads (UMS)
 - В 64 битной версии
 - Есть контекст потока в ядре, поэтому можно получить управление при блокирующем вызове потока, можно использовать несколько процессоров
- Asynchronous Procedure Call (APC) и Deffered Procedure Call (DPC)

Fiber:

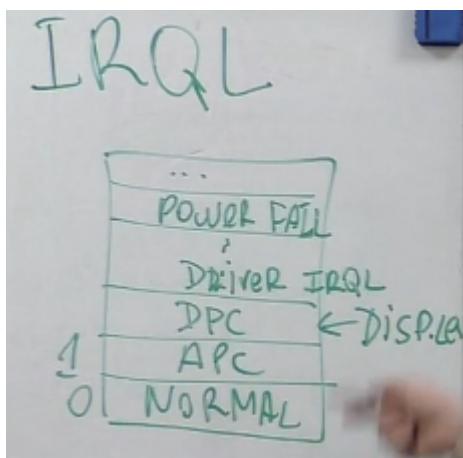
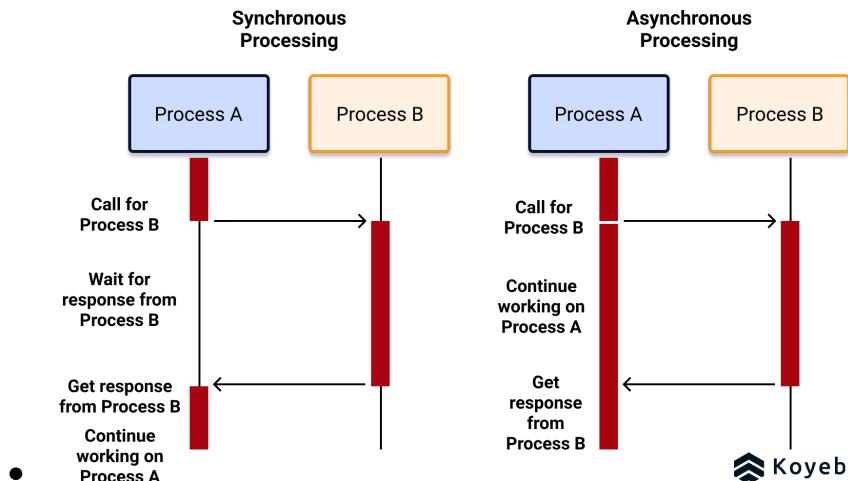
- Обычно предоставляют собой соответствие много Fiber к одному диспетчеризируемому потоку.
- Реализуются на базе Kernel32.dll
- Cooperative multitasking - подразумевает то, что поток сам освобождает ресурсы процессора для следующего потока.
- Если 64-битная система, то рекомендуется заменить на UMS

UMS:

- Отличается тем, что можно создать несколько контекстов ядра и назначить на них некоторое количество потоков. (могут работать параллельно на нескольких процессорах)

APC:

- Выполняются на уровнях прерывания ядра больших чем пользовательский режим. Обычно выполняются в контексте треда, который диспетчеризуется в ядре. То есть выбирается тред, к нему подвешивается callback и в определенный момент времени он исполняется, например, когда thread заканчивает свой квант времени.



IRQL (IRQ level) - программный уровень прерывания ядра, в котором будут выполняться различные функции ядра. Когда вызывается прерывание, ему назначается уровень прерывания ядра и в отличие от уровня поток тем или иным образом диспетчеризуется на процессе.

Подробнее/другими словами из столлингса:

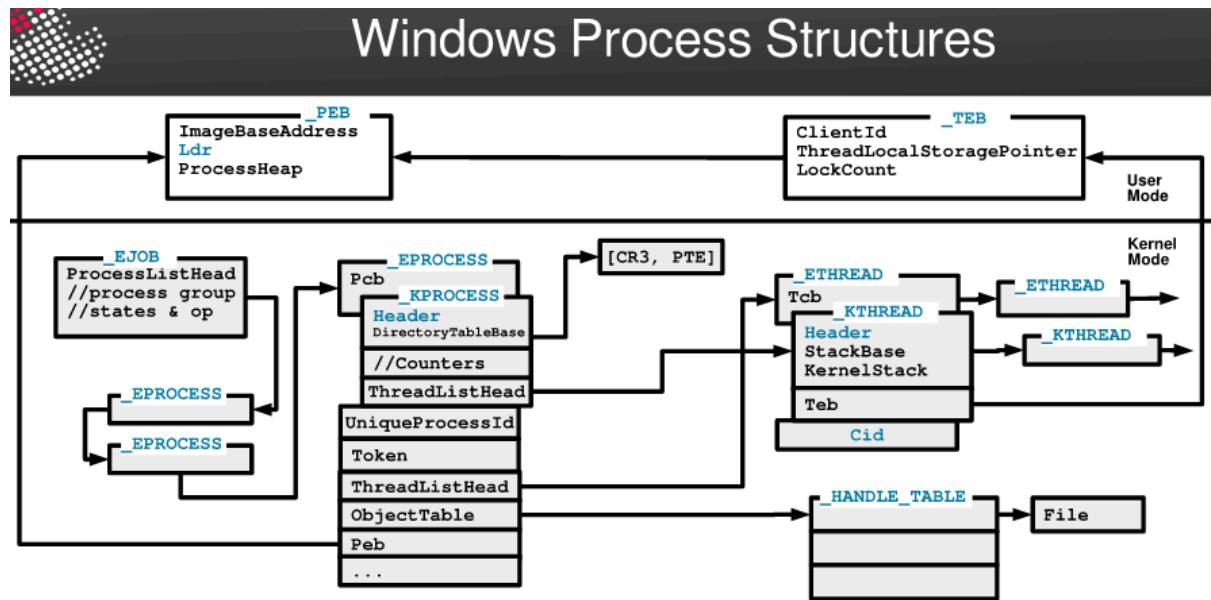
Приложение состоит из одного или нескольких процессов. Каждый процесс представляет ресурсы, необходимые для выполнения программы. Процесс имеет виртуальное адресное пространство, выполнимый код, открытые дескрипторы системных объектов, контекст безопасности, уникальный идентификатор процесса, переменные среды, класс приоритета, минимальный и максимальный размеры рабочего множества и по крайней мере один поток выполнения. Каждый процесс запускается с одним потоком, который часто называют основным или первичным, но любой из его потоков может создавать дополнительные потоки.

Поток является сущностью в рамках процесса, которая может быть запланирована к выполнению. Все потоки процесса разделяют его виртуальное адресное пространство и системные ресурсы. Кроме того, каждый поток поддерживает обработчики исключений, приоритет планирования, локальную память потока, уникальный идентификатор потока и набор структур, которые система будет использовать для сохранения контекста потока до тех пор, пока он не будет запущен планировщиком. В многопроцессорном компьютере система может одновременно выполнять столько потоков, сколько имеется процессоров.

Волокно (fiber) — это единица выполнения, которая должна планироваться приложением вручную. Волокна выполняются в контексте потоков, которые их планируют. Каждый поток может планировать несколько волокон. В общем случае волокна не предоставляют преимущества по сравнению с хорошо спроектированным многопоточным приложением. Однако применение волокон может упростить перенос приложений, которые были разработаны с применением планирования собственных потоков. С точки зрения системы волокно является тождественным потоку, который его выполняет. Например, если волокно обращается к локальной памяти потока, оно обращается к локальной памяти того потока, которые его выполняет. Кроме того, если волокно вызывает функцию `ExitThread`, то поток, который его выполняет, завершает свою работу. Однако волокно не имеет всей той же информации о состоянии, связанной с ним, что и информация, связанная с потоком. Единственная поддерживаемая волокном информация о состоянии — его стек, подмножество его регистров и данные волокна, предоставленные при его создании. Сохраненные регистры представляют собой набор регистров, обычно сохраняемых при вызове функции. Волокна не подлежат вытесняющему планированию. Поток выполняет планирование, переключаясь на одно волокно с другого. Система же по-прежнему занимается планированием выполнения потоков. Когда прерывается поток, выполняющий волокна, выполнение текущего волокна прерывается, но оно остается выбранным.

Пользовательский режим планирования (*user-mode scheduling* — UMS) представляет собой облегченный механизм, который приложения могут использовать для планирования собственных потоков. Приложение может переключаться между потоками UMS в пользовательском режиме без участия системного планировщика и восстанавливать контроль над процессором, если поток UMS блокируется в ядре. Каждый поток UMS имеет собственный контекст вместо совместного использования контекста единственного потока. Возможность переключения между потоками в пользовательском режиме делает UMS более эффективным, чем пул потоков для краткосрочной работы, которая требует нескольких системных вызовов. UMS полезен для приложений с высокими требованиями к производительности, которые должны эффективно выполнять несколько потоков одновременно на многопроцессорных или многоядерных системах. Чтобы воспользоваться преимуществами UMS, приложение должно реализовать компонент планировщика, который управляет UMS-потоками приложения и определяет, когда они должны быть выполнены.

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



Данное описание не является чисто виндовское, а получено в результате реверса.

Центральной структурой является **_EPROCESS** (Effective Process).

- Header - заголовок DispatcherObject (96 билет)
- UniqueProcessId - идентификатор процесса
- Token - токен безопасности
- ThreadListHead - список потоков, принадлежащих процессу
- ObjectTable - содержит таблицу с объектами (**_HANDLE_TABLE**), которые принадлежат процессу (большинство системных сущностей представлены в виде объектов). Указывает на объекты определенных типов. В примере на слайде - указывается на файлы.

В **_EPROCESS** встроена структура **_KPROCESS** (Kernel Process):

- Хранит информацию, которая больше связана с исполнением процесса на процессоре.
- DirectoryTableBase - содержит пару значений, которая указывает на начало виртуальной таблицы преобразований адресов (CR3 - содержимое регистра cr3, PTE - page table entry)
- Различные счетчики производительности

Вот структура **_KPROCESS** обычно находится в начале **_EPROCESS** и называется **Pcb** - process control block.

ETHREAD:

- Tcb - по аналогии с процессами. Это **_KTHREAD**. В нем в основном описаны стеки (`StackBase`, `KernelStack`).

Все это находится на Kernel Mode. Для того, чтобы получить доступ к частям User Mode есть объекты **Peb** (Process environment block). В нем находятся описания

процесса, ссылки на важные части: на загруженные библиотеки, на процессорный heap и т.д.

Аналогичная структура TEB. Там находятся локальные данные для каждого потока, регистры, которые перегружаются при переключении контекста.

ClientId - совокупность идентификатора процесса и идентификатора потока.

EJOB - соответствует jobам:

- Список процессов, которые этому jobу принадлежат.
- Также описывает состояния и операции с этой группой процессоров
- ProcessListHead - входящие процессы

Также есть глобальные списки процессов, которые пронизывают все процессы и все потоки.

Дополнительно:

CSRSS - client server runtime subsystem. Этот процесс на user mode содержит структуры, которые частично зеркалируют эту информацию для клиент-серверной подсистемы. Нужно для более эффективной диспетчеризации если ос для клиент-серверного взаимодействия.

- Поля **_EPROCESS**

- PCB
- Защищающие блокировки
- UniqueProcessID
- Ссылки списка процессов (начало в PsActiveProcessHead)
- Флаги
- Времена создания и завершения
- Информация о квотах
- Ссылка на сессию
- Основной токен доступа
- Ссылка на задание
- Объекты процесса (Handle Table)
- Окружение процесс (PEB)
- Имя фала образа процесса
- Счетчики производительности
- Список потоков
-

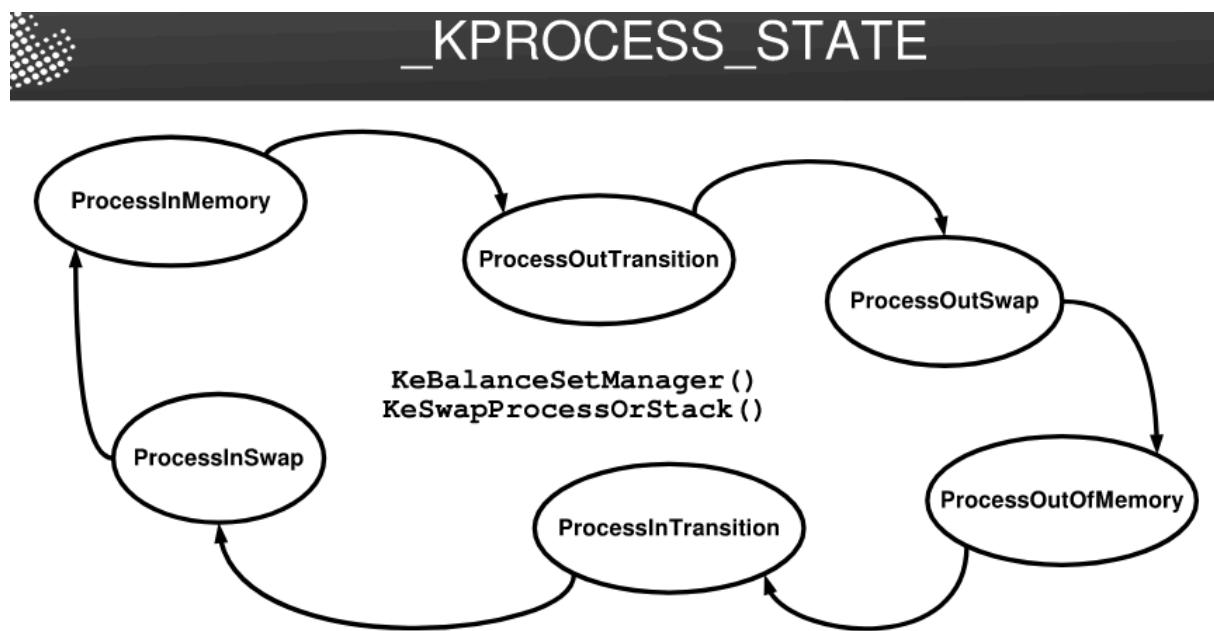
- Поля **_KPROCESS**

- Win32K структура
- WOW64 структура
- Pico Context
- DirectX process
- Набор рабочих страниц WorkingSet
- Секции (сегменты) образа
- Заголовок диспетчера
- Ссылка на таблицу страниц
- Kernel/User/Cycle Times
- Context Switches
- Список Thread
- Аффинити
- Флаги
- Базовый приоритет

Аффинити - принадлежность процессора к какому-то процессу.

- Поля _ETHREAD
 - TCB
 - Время создания и завершения
 - Ссылка на процесс
 - Флаги потока
 - Токен доступа
 - Стартовый адрес
 - Ожидаемые запросы ввода-вывода
 - Таймеры
 - Код выхода
 - Потребление энергии
 - Процессорный набор
- Поля _KTHREAD
 - Заголовок диспетчера
 - ТЕВ
 - Kernel/User Time
 - Указатель на SSDT (system services descriptor table)
 - Freeze/Suspend Count
 - Win32 Thread Object
 - Процесс (_KPROCESS)
 - Информация о стеках
 - Информация планировщика
 - Фрейм прерывания
 - Информация о синхронизации
 - Информация о wait
 - Список объектов ожидания для потока
 - Список APC, ожидающих обработки

94. Диаграммы состояний процесса и потока Windows



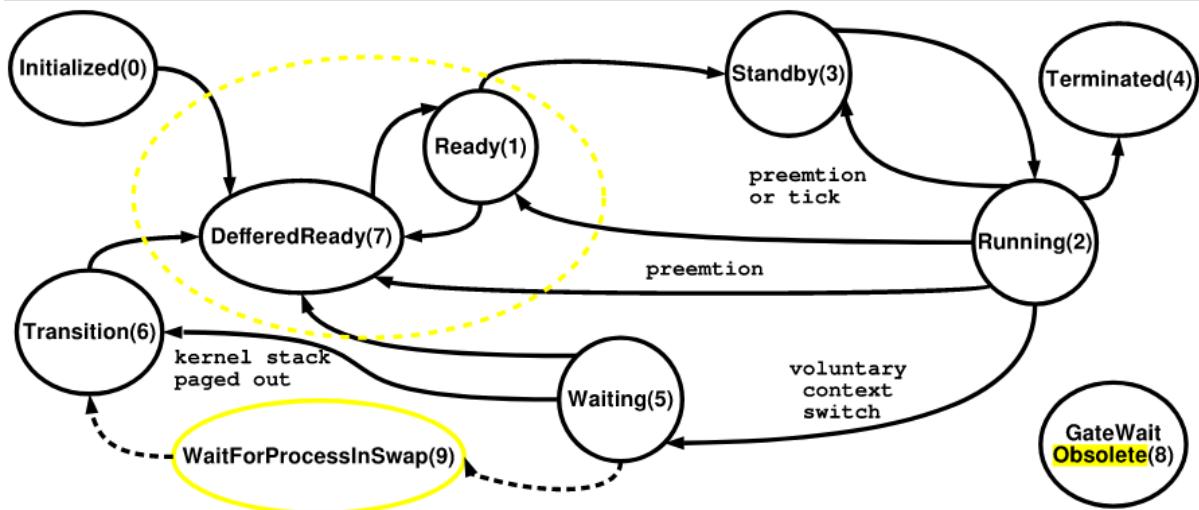
У процесса есть несколько полей, которые его определяют его состояние. Основным состоянием является _KPROCESS_STATE. Относится к долговременному планированию процесса и BalanceSet (взаимодействует с областью подкачки).

KeBalanceSetManager() - управление с областью подкачки
 KeSwapProcessOrStack() - загрузка и выгрузка процесса.

Состояния процесса:

- ProcessInMemory - процесс в памяти
- ProcessOutOfMemory - процесс в свопе
- ProcessOutTransition - началась выгрузка с определенных страниц (помечен для выгрузки)
- ProcessOutSwap - процесс выгружается
- ProcessInTransition - поставлен в очередь на диспетчеризацию загрузки
- ProcessInSwap - процесс загружается

_KTHREAD_STATE



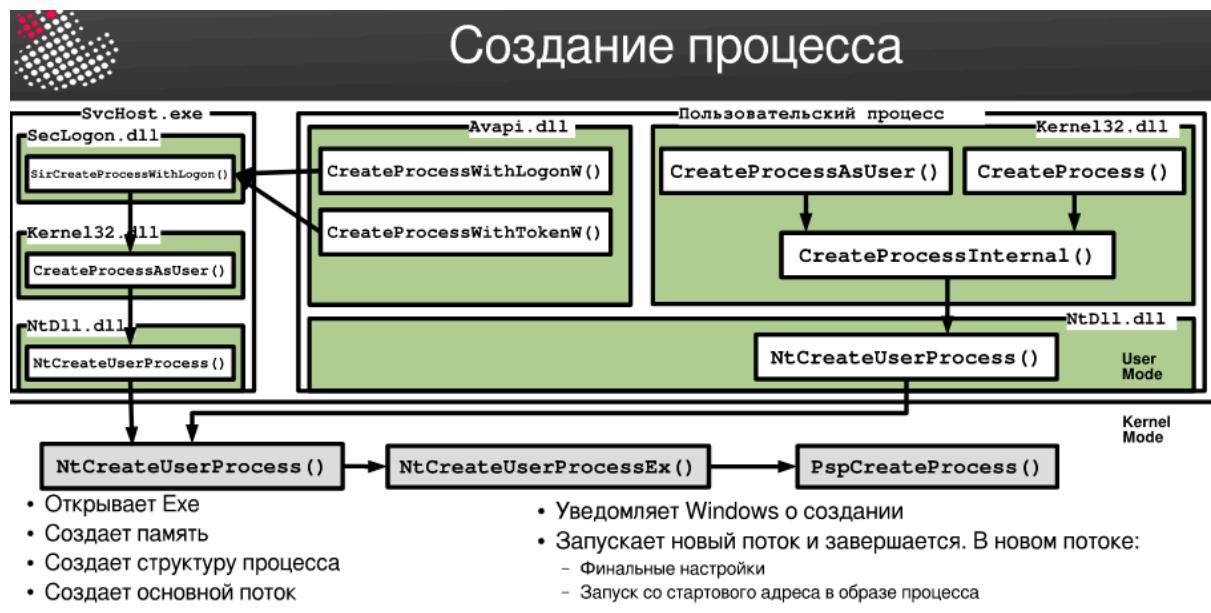
Операционные системы. Часть 2. Программные потоки

Единицей диспетчеризации является поток.

Состояния потока:

- Initialized - поток создан
 - Terminated - поток прерван
 - Waiting - состояние ожидания
 - Running - поток находится на процессоре
 - DefferedReady - процессор, на котором будет выполняться поток еще не выбран
 - Ready - в этом состоянии выбран процессор, на котором выполняется поток
 - StandBy - тут находится тот процесс, который будет выполняться следующим
 - Transition - попадает из waiting (ожидается пока процесс не будет полностью загружен, критически важные страницы)
 - WaitForProcessInSwap - промежуточное состояние (помечено что процесс должен быть загружен)
 - GateWait (Obsolete) - процесс ожидает gate (примитива синхронизации)

95. Создание и завершение процесса Windows.



С UserMode осуществляется NtCreateUserProcess(), потом выполняется syscall. NtDII.dll - библиотека системных вызовов нижнего уровня. К ней подключаются библиотеки, которые может использовать пользователь.

В Kernel32.dll находится функция CreateProcess() и CreateProcessAsUser() и ниже понятно.

Avapi.dll:

- CreateProcessWithLogonW() - создание процесса с помощью логина
- CreateProcessWithTokenW() - создание процесса с помощью токена безопасности
- Использует SvcHost.exe по коммуникации (удаленной). Чтобы можно было удаленно запускать процессы удаленно.

Kernel:

- NtCreateUserProcess() - обертка, проверяющая несколько флагов
- NtCreateUserProcessEx() - тоже какая-то небольшая
- PspCreateProcess() - уже создает все необходимое (описание что он делает снизу слайда)



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

TerminateProcess() - с проверкой прав

Оповещение DLL - чтобы библиотеки освободили необходимые ресурсы (файлы, сетевые соединения)

handles - объекты таблицы handle

96. Примитивы синхронизации Windows. Понятие Dispatcher Object.

Ожидание наступление события, вызовы Wait.

<https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel>



Kernel Object = { Dispatcher Object
Control Object }

- Любой объект ядра, на котором можно ожидать события - «dispatcher object»
 - Некоторые предназначены только для синхронизации (events, mutexes, semaphores, queues, ...)
 - Другие для ожидания событий от процессов, потоков, файлов
- Общая структура данных
- Два состояния
 - «Signaled» - ожидание удовлетворено и «Not-signaled»
 - еще нет.
 - Различные объекты отличаются в том, что именно изменяет их состояние
 - Wait и unwait — операции общие.

```
ntos/inc/ntosdef.h
typedef struct _DISPATCHER_HEADER {
    union {
        struct {
            UCHAR Type;
            union {
                UCHAR Absolute;
                UCHAR NpxIrql;
            };
            union {
                UCHAR Size;
                UCHAR Hand; B win10  
больше!
            };
            union {
                UCHAR Inserted;
                BOOLEAN DebugActive;
            };
        };
        volatile LONG Lock;
    };
    LONG SignalState;
    LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;
```

Любой объект ядра - Dispatcher Object или Control Object.

Dispatcher object - любой объект ядра, на котором можно ожидать появления события (диспетчер знает об этом объекте).

Управляет одной структурой `_DISPATCHER_HEADER` (справа) - хедер объекта диспетчера.

- Type - тип примитива синхронизации + во внутренних unionах - информация, соответствующая этому типу. Покрывается Lock (потому что union).
- SignalState - состояние объекта диспетчера (Signalled -> 0). Типы событий, которые вызывают этот state на следующем слайде, но вообще зависят от типа объекта.
- WaitListHead - голова очереди потоков, которые ожидают этот объект.

События ожидания реализуются для всех объектов синхронизации одними и теми же функциями ожидания (`wait`, `unwait`).

Пример: thread прекратил свою работу - потоки, которые ожидают прекращения этого треда будут автоматически оповещены диспетчером.

Вызовы Wait

- Гибкие вызовы ожидания — `KeWaitForSingleObject`, `KeWaitForMultipleObjects`
 - Ожидает одного или нескольких объектов ("any" или "all"). В случае всех — все объекты должны быть в состоянии «signaled»
 - Таймаут задается опционально
- Объекты, которые можно ожидать:
 - Events, Mutexes, Semaphores, Timers
 - Processes and Threads (exit or terminate)
 - Directories (change notification)
- Нет гарантированного порядка завершения ожидания
 - Потоки ожидающие события пробуждаются в неопределенном порядке, зависит от диспетчера
 - Обычный порядок - FIFO; Однако APC может изменить этот порядок

```
ntos\inc\ke.h
typedef enum _KWAIT_REASON {
    Executive,
    FreePage,
    PageIn,
    PoolAllocation,
    DelayExecution,
    Suspend,
    Suspended,
    UserRequest,
    IrpExecutive,
    IrpFreePage,
    IrpPageIn,
    IrpPoolAllocation,
    IrpSuspend,
    IrpSuspended,
    IrpUserRequest,
    IrpEventPair,
    IrpQueue,
    IrpIpcReceive,
    IrpIpcReply,
    IrpVetoedMemory,
    IrpPageOut,
    IrpRendezvous,
    Spare2,
    Spare3,
    Spare4,
    Spare5,
    Spare6,
    IrpKernel,
    IrpResource,
    IrpPushLock,
    IrpMutex,
    IrpQuantumEnd,
    IrpDpcSchtInt,
    IrpPreempted,
    IrpYieldExecution,
    IrpFastMutex,
    IrpGuardedMutex,
    IrpRundown,
    MaximumWaitReason
} KWAIT_REASON; RE
```

Операционные системы. Часть 2. Процессы и потоки
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020



Диспетчер учитывает не только постановку в очереди, но и приоритет, именно поэтому нет гарантированного порядка завершения ожидания.

- APC - асинхронный вызов процедур.

Справа перечислены все причины, по которым можно ожидать события.

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

Все вышеописанное является Dispatcher Kernel Object.



EventObject

- Может сбрасываться вручную, автоматически и «pulsed»
- Используется для оповещения о наступлении события двух типов:
 - Синхронизация — один из ждунов продолжает выполнение когда наступает состояние Signaled. EO автоматически сбрасывается в Not-Signaled
 - Нотификация — пробуждает всех ждунов, требует ручного сброса состояния
- Операции:
 - KeInitializeEvent - Initialize an event object
 - KePulseEvent - Set/reset event object state atomically
 - KeReadStateEvent - Read state of event object
 - KeResetEvent - Set event object to Not-Signaled state
 - KeSetEvent - Set event object to Signaled state

```
ntos/inc/ntosdef.h
typedef struct _KEVENT {
    DISPATCHER_HEADER Header;
} KEVENT, *PKEVENT, *PRKEVENT;
```

По сути не содержит ничего кроме хедера DISPATCHER_HEADER.

- Сброс вручную: дождались события, необходимо вызвать процедуру который сбрасывает состояние ожидания.
- Pulsed - “мы его взводим и оно сразу сбрасывается, типа импульса”.

В основном используются для ожидания в общем смысле наступления какого-либо системного события: один поток это событие триггерит, а другие несколько потоков будут это состояние ожидать.



Mutexes и Mutants

- Mutually exclusive, deadlock free доступ к разделяемым ресурсам
- В нормальном режиме создается в «signaled» состоянии
- Возможен рекурсивный захват (сколько захватов, столько освобождений)
- Захваченный mutex блокирует выход из Kernel Mode
- Снятие Lock — mutant: любой поток (abandoned state); mutex: владелец
- В mutex запрещены APC, в mutant — нет.
- Возможно использовать mutant в UserLand
- Increment — добавляется к приоритету потока если wait satisfied
- Wait — за KeRelease* сразу следует функция ожидания (освободить и занять в рамках «атомарной операции»)

```
ntos/inc/ke.h
typedef struct _KMUTANT {
    DISPATCHER_HEADER Header;
    LIST_ENTRY MutantListEntry;
    struct _KTHREAD *OwnerThread;
    BOOLEAN Abandoned;
    UCHAR ApcDisable;
} KMUTANT, KMUTEX;
```

- Mutant:
 - KeInitializeMutant
 - KeReadStateMutant
 - KeReleaseMutant
 - * KPRIORITY Increment,
 - * BOOLEAN Abandoned,
 - * BOOLEAN Wait
- Mutex:
 - KeInitializeMutex
 - KeReadStateMutex
 - KeReleaseMutex
 - * KeReleaseMutant(mutex, 1, FALSE, Wait)

Раньше в винде были мьютексы и мутанты, но со временем мьютекс стал реализован через основные функции поддержки мутантов. Эти примитивы имеют общую структуру _KMUTANT.

Состоит из:

- Header - заголовок
- OwnerThread - поток, который захватил блокировку
- ApcDisable (асинхронный вызов процедур) - разрешен или запрещен
- Abandoned - является ли блокировка покинутой.

Если используются мутанты в UserLand, то они как бы одновременно находятся и на стороне ядра и на стороне пользователя.

Когда ожидание заканчивается, то необходимо поднять приоритет процесса, чтобы событие было быстрее обработано.

Wait - необходимо, чтобы диспетчер выполнил операцию как одно целое, чтобы не произошло диспетчеризации и процесс не уехал на другой процессор.

Логика работы:

1. Создается мьютекс в состоянии signalled.
2. Первый wait попадет в критический участок и при этом можно будет эту блокировку захватить (signalled = 0).
3. Следующий wait будет ожидать signalled.

Освобождение блокировки происходит через ReleaseMutex.

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



Fast mutexes и Guarded mutexes

- Больше похожи на «нормальные» мьютесы
- Поле Count: 0-й бит — lock, 1-й — single waiter woken
- Нельзя захватывать рекурсивно
- Операции:
 - ExInitializeFastMutex
 - ExAcquireFastMutex
 - ExTryToAcquireFastMutex
 - ExReleaseFastMutex
- До Windows 8 — разные реализации, после - идентичны

```
ntos/inc/ex.h
typedef struct _FAST_MUTEX {
    LONG Count;
    PKTHREAD Owner;
    ULONG Contention;
    KEVENT Event;
    ULONG OldIrql;
} FAST_MUTEX, *PFAST_MUTEX;
```

Рекурсивный мьютекс — это особый тип устройства взаимного исключения (мьютекса), которое может быть заблокировано несколько раз одним и тем же процессом/потоком, не вызывая взаимоблокировки. Главное чтобы блокировок и разблокировок было одинаковое количество.

Ожидание мьютекса происходит локально по отношению к процессору.

Поля:

- Count - количество ожидателей. 0 бит - lock, 1 - signal waiter woken.
- Owner - тот, кто захватил мьютекс.
- Contention - есть ли состязание за захват
- oldIrql - уровень прерываний

Guarded mutex - алиас fast mutex.

99. Примитивы синхронизации Windows. Semaphore, spinlock



Semaphores

- Управляют захватом разделяемого ресурса
 - Limit — максимальное количество такого ресурса
- Count — начальное состояние, помещается в Header.SignaledState
- Семафор открыт, когда SignaledState > 0
- Если SignaledState+Adjustment > Semaphore→Limit
 - вызывается ExRaiseStatus(STATUS_SEMAPHORE_LIMIT_EXCEEDED)
- Increment — добавляется к приоритету потока если wait satisfied
- Wait — за KeReleaseSemaphore сразу следует функция ожидания (освободить и занять в рамках «атомарной операции»)

```
ntos/inc/ke.h
typedef struct _KSEMAPHORE {
    DISPATCHER_HEADER Header;
    LONG Limit;
} KSEMAPHORE;
```

- KeInitializeSemaphore
 - LONG Count
 - LONG Limit
- KeReadStateSemaphore
- KeReleaseSemaphore
 - KPRIORITY Increment,
 - LONG Adjustment,
 - BOOLEAN Wait

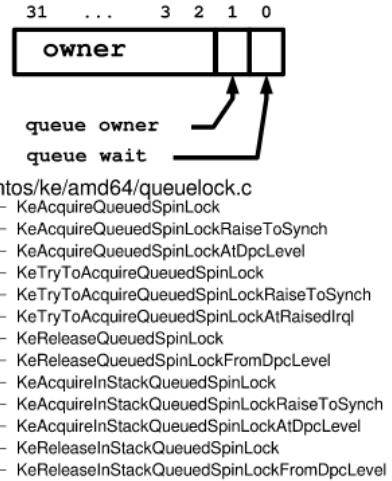
Точно также как и мьютексы создаются в signaled state. Описание дублируется с тем, что выше.

- SignaledState + Adjustment > Semaphore->Limit (если значение свободного ресурса превышает максимально допустимое, то вызывается исключение ExRaiseStatus).



Spinlock (Queued Spinlock)

- Используется атомарная операция test-and-modify
 - Синхронизация на многопроцессорных системах при помощи локальной переменной
- Один владелец в один момент времени
- Реализация - одна ячейка памяти
 - `typedef ULONG_PTR KSPIN_LOCK;`
- Для Windows > XP рекомендуется использовать `AcquireInStack`



<https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-spin-locks>

Операционные системы. Часть 2. Процессы и потоки

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-it LTD 1999-2020

tuneit 104

Также как и на линуке основное предназначение - быстрая обработка при активном ожидании, да и устроен похоже.

- Есть очереди и разные локальные переменные, чтобы снизить bus contention, когда атомарная операция блокирует шину при опросе блокировки.
- Младшие биты также используются, чтобы пометить что есть ожидание на данной блокировке.

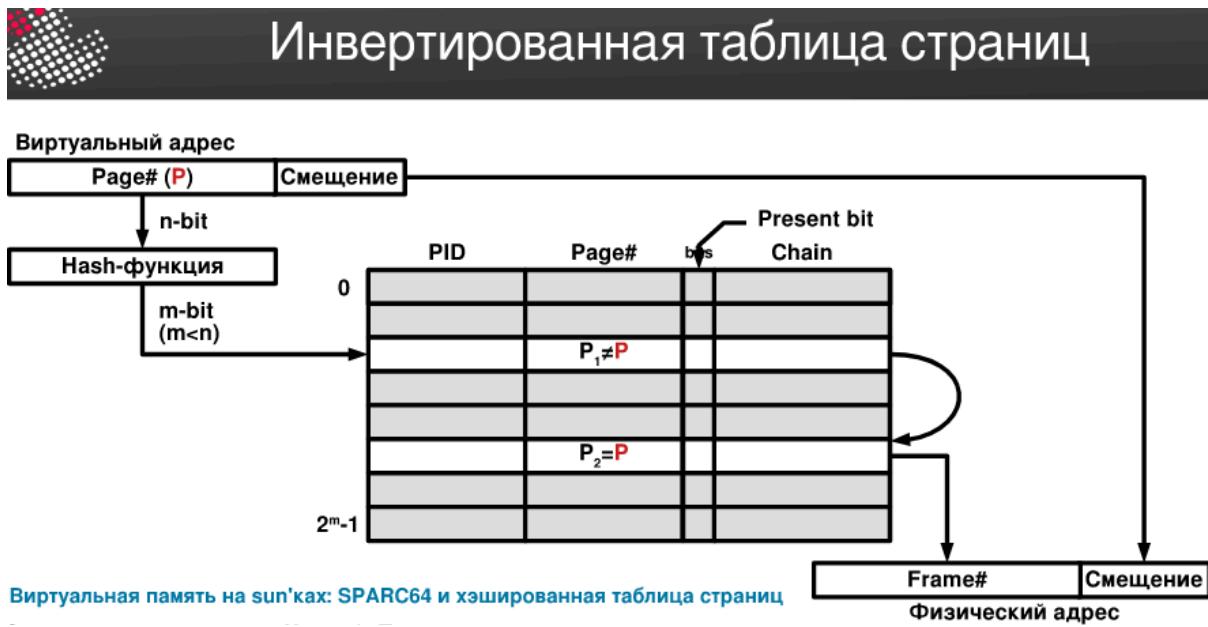
Ссылки, что гроб не казался гробом (хотя там ничего полезного нет):

<https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-spin-locks>

Интересный факт оттуда же:

- Спинлоки довольно часты в драйверах.
- Во время спинлока повышается уровень Irql, чтобы другой процесс не мог быть запланирован на цпу.

100. Хешированная таблица страниц SPARC64.

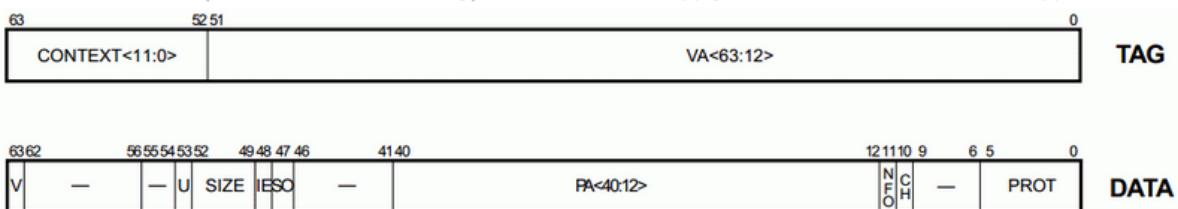


В простейшем случае (не спарк) оно выглядит так. Как это работает:

На входе есть n-битный адрес. Он попадает в hash-функцию, и в результате получаем m-разрядный адрес, который используется как индекс в таблице. В таблице хранятся записи о виртуальных адресах, mapping'и которых существуют. Далее происходит сравнение mapping'ов, которые существуют с заданным. Маппинги в одном "бакете" связаны chainом. Затем по chain'у как linked list мы проходимся и находим нужную нам запись физического адреса. Если в этой ячейке нолик, то других связанных записей нет.

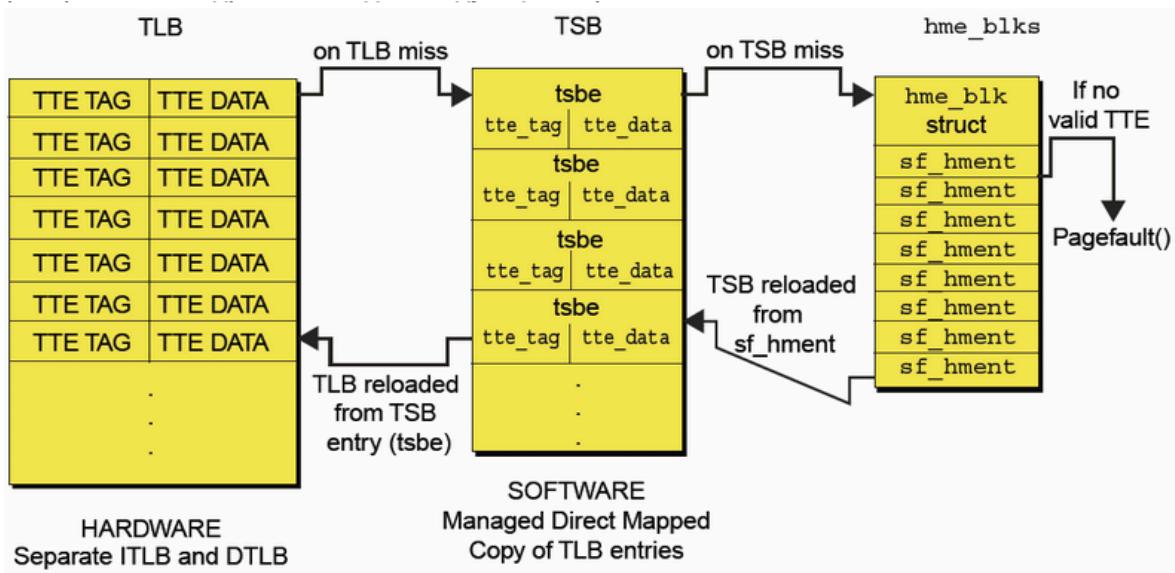
Плюсы данной реализации: сокращение таблицы mapping'ов.

Реализация SPARC64:



Отдельная запись в кеше определяет расположение одной виртуальной страницы в физической памяти и называется TTE (Translation Table Entry). TTE состоит из тега и данных. Размер и того и другого - 64 бит.

Поиск осуществляется по тегу, состоящему из номера контекста и номера виртуальной страницы. При совпадении используется информация из Data - номер физической страницы (VA), и т.д.



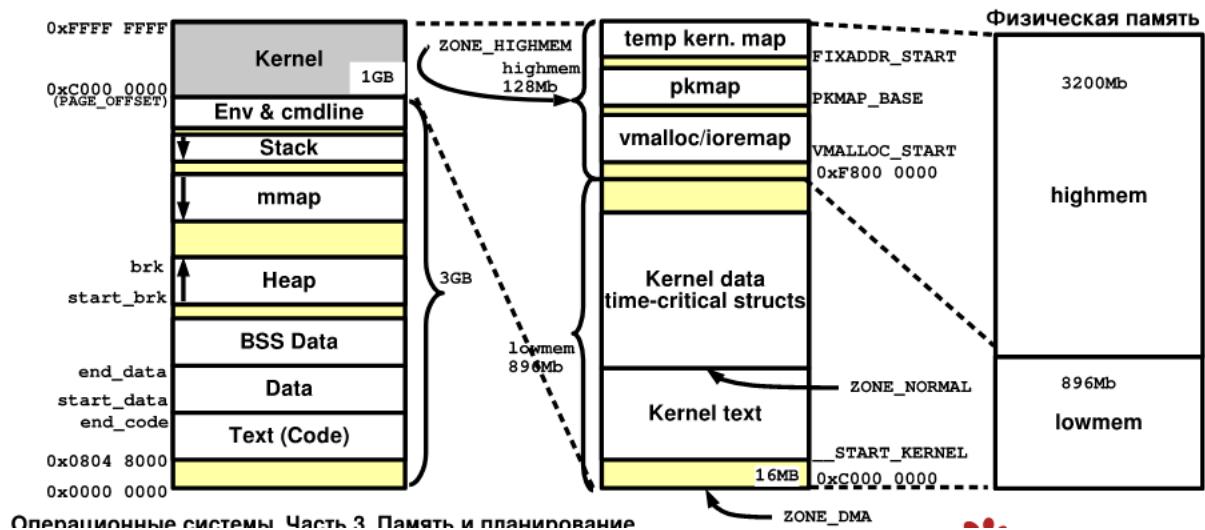
В случае TLB промаха, управление передается в ОС соответствующему обработчику прерывания. ОС может делать что угодно для поиска физ адреса, однако разумно предположить, что сначала она посмотрит в специальную структуру TSB (Translation Storage Buffer).

Для каждого процесса существует свой набор таких структур. Кроме этого у каждого процесса имеется таблица TSB, располагающаяся в основной памяти и используемая в качестве программно-управляемого кэша. Туда помещается ограниченное количество записей о расположении виртуальных страниц. Одна строка TSB занимает 128 бит и по содержимому практически совпадает с TTE, хранящемуся в TLB.

Набор структур **hme_blks** целиком описывает одно виртуальное адресное пространство и организован в виде [хэшированной таблицы с цепочками](#).

После TLB промаха, ОС попытается найти запись о запрошенном виртуальном адресе в TSB. Если запись найдена, то ОС загрузит её в TLB, иначе попытается найти её в **hme_blks**.

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



Операционные системы. Часть 3. Память и планирование

Любой процесс, который запущен, состоит из сегментов. Начало памяти (снизу) стараются делать неразмапленным, чтобы код, который работал в старых операционных системах, чтобы он вызывал page fault и эта память была недоступна. До 0x08048000 - память неразмаплена.

Сегменты:

- Text (code) - тут код
- Data - тут данные
- BSS (zero set segment) Data - статические неинициализированные или инициализированные нулями данные.

Heap:

Начинается с start_brk, растет до brk (program break) вверх. Malloc как правило (если не реализован через mmap) двигает указатель brk.

Mmap:

Раньше рос вверх, сейчас вниз. Туда загружаются библиотеки или когда мы вызываем syscall mmap.

Если mmap и heap слишком большие, то чтобы данные не перехлестнулись, то получим ENOMEM.

Stack:

Растет вниз. За стеком сверху (вроде бы) находится защитная область памяти

Env & cmdline:

Там хранятся переменные окружения.

На kernel:user память делится 1:3.

Ядро:

Делится на 3 зоны: ZONE_DMA, ZONE_NORMAL, ZONE_HIGHMEM.

ZONE_DMA: находятся области памяти для старых устройств (16МБ).

ZONE_NORMAL: содержит сегмент кода ядра (512МБ).

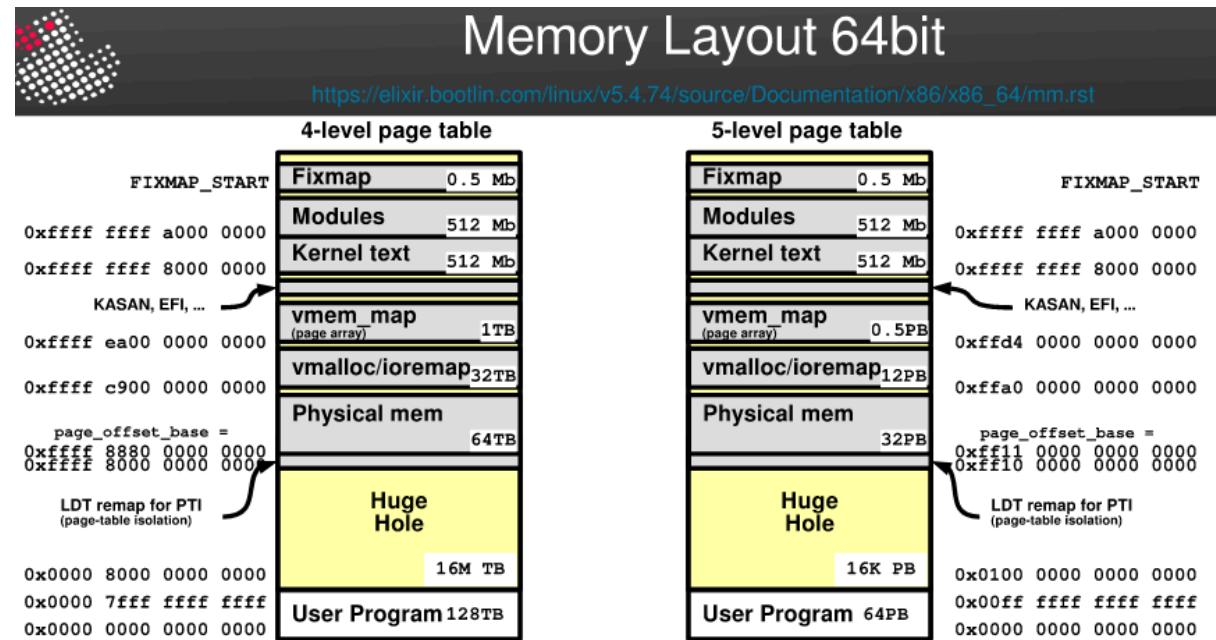
ZONE_HIGHMEM: до конца.

До 896МБ находится область памяти lowmem, которая один к одному мапится в нулевые адреса. Поэтому нужна только одна запись в TLB.

highmem: Для них можно создать записи в highmem и переключать страницы по необходимости. Там можно выделять динамическую память (vmalloc), маппинги для ввода-вывода (ioremap), pkmap - интерфейс подключения и отключения страниц по необходимости, temp kern map - временные kernel маппинги, чтобы не было подгрузки страниц и т.п.

Ядро может видеть контекст процесса через регистр cr3, который указывает на таблицу страниц.

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

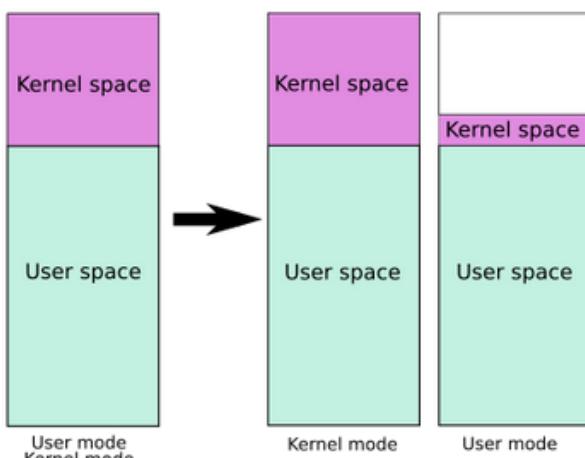


Следует обратить внимание, что слева четырехуровневая (47 бит), а справа пятиуровневая таблица страниц (56 бит).

Huge hole: обусловлена невозможностью размапить эти адреса, так как физической памяти мало. Поэтому принято решение расширить старший знак. В связи с этим данный кусок памяти не может быть никак размаплен. В четырехбитной - 16 миллионов терабайт, в пяти - 16 тысяч петабайт. Дальше начинается ядро.

Page-table isolation - для защиты ядра. Из пользовательского пространства системных вызовов нельзя получить к памяти доступ ко всей памяти ядра.

Kernel page-table isolation



Подробнее: КПТИ устраняет эти утечки, полностью разделяя таблицы страниц пространства пользователя и пространства ядра. Один набор таблиц страниц включает адреса пространства ядра и пространства пользователя, как и раньше, но он используется только тогда, когда система работает в режиме ядра. Второй набор таблиц страниц для использования в пользовательском режиме содержит копию пользовательского пространства и минимальный набор отображений пространства ядра, который предоставляет информацию, необходимую для входа или выхода из системных вызовов, прерываний и исключений.

Так как адресное пространство очень большое, то вся физическая память последовательно мапится в пространство ядра начиная с `page_offset_base`. То есть для того, чтобы получить доступ, нужно взять номер физического фрейма, прибавить к `page_offset_base` и получить нужный виртуальный адрес.

`vmalloc/ioremap` - область динамической аллокации.

`vtmem_map` - там находятся структуры, описывающие страницы. То есть если ядру нужно создать страницу, то оно все описательные структуры положит туда.

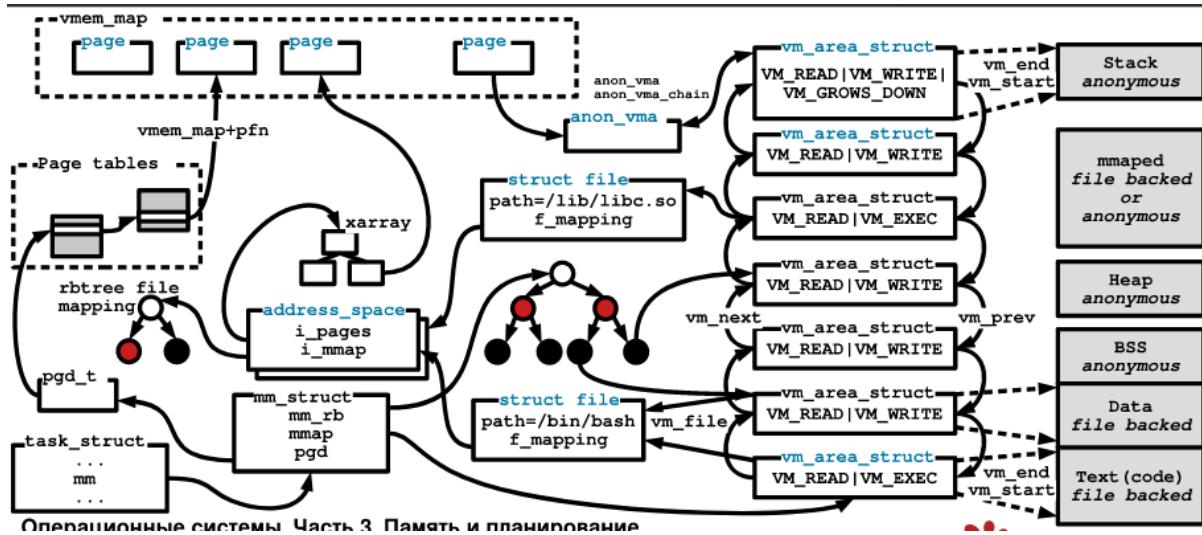
Дальше находится куча всего: KASAN, EFI, доп. отладочная информация.

`Kernel text` - код ядра

`Modules` - модули ядра.

`Fixmap` - временная таблица для временных маппингов, которые не вызывают блокировки.

103. Виртуальная память Linux. Структуры памяти.



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. М 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 страницке: **READ**, **EXEC**; и есть указатели на следующий и предыдущий сегмент, чтобы в случае необходимости их можно было подвернуть изучению).

В правом «столбце» в явном виде под каждым сегментом подписано, как бэкендятся данные. Т.е. условно говоря, что «лежит под» этим самым сегментом. Имеется в виду, что сегменты «внизу себя» хранят либо свой-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 страницки **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-чёрное 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.

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

В **vtmem_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 страницкой 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 страницки, которые временно скэнированы 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-чёрное дерево, которое устанавливает связи с блоками данных, которые являются файлом.

104. Способы выделения памяти для пользовательских процессов

Linux



Выделение памяти user-space

- **Malloc**

- Несколько реализаций (buddy, glibc, ...)
- При нехватке памятидвигает program break — sbrk()

- **Mmap**

- Гибкий вызов создания областей памяти
- Создает «file backed» и анонимную память

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void *reallocarray(void *ptr,
                  size_t nmemb, size_t size);

void *mmap(void *addr, size_t length,
           int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);

PROT_NONE      MAP_SHARED      MAP_LOCKED
PROT_EXEC      MAP_PRIVATE     MAP_NONBLOCK
PROT_READ       MAP_32BIT       MAP_NORESERVE
PROT_WRITE      MAP_ANONYMOUS   MAP_POPULATE
                  MAP_FIXED        MAP_STACK
                  MAP_GROWSDOWN  MAP_UNINITIALIZED
                  MAP_HUGETLB
                  MAP_HUGE_2MB
                  MAP_HUGE_1GB
```

sbrk() - подвинуть program break. Довольно простая операция по сравнению с mmap.

Если в linux выделяется большое количество памяти, то вместо типичного sbrk() выполняется mmap (детали реализации для улучшения производительности: что быстрее создать новые маппинги памяти через mmap или подвинуть program break через sbrk(), который конечно тоже создает новые маппинги, но они будут в новом page table, значит быстрее).

Справа снизу перечислены опции, чтобы установить особенности создаваемого сегмента.

prot - желаемая защита памяти. (PROT_NONE, PROT_EXEC, PROT_READ, PROT_WRITE).

flags - MAP_*. Так mmap может создавать shared memory между процессами, заблокированные страницы, которые не будут участвовать в paging, swapping, создавать большие странички и так далее.

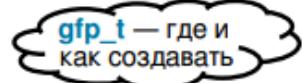
105. Способы выделения памяти в пространстве ядра Linux.



Выделение памяти в пространстве ядра

- Выделение страниц

- `page *alloc_pages(gfp_t flags,order)` — выделяет 2^{order} последовательных страниц в памяти, возвращает `struct page`; `free_pages()` - освобождение; `_get_free_pages(gfp_t flags ,order)` - возвращает адрес

 `gfp_t` — где и как создавать

- Динамическая аллокация

- `vmalloc(unsigned long size), vzalloc()` - выделение непрерывной области из нескольких страниц, достаточных для `size`; `vmalloc_user()` - выделение обнуленный страниц в пространстве пользователя
- `ioremap(), iounmap()` - мапинг области ввода-вывода в физической памяти в область виртуальной памяти
- `page_frag_alloc(), page_frag_free()` - выделение фрагментов страниц, оптимизация для сетевых драйверов
- `kmalloc(size, gfp_t flags), kfree()` — выделение памяти меньшей чем `page` в различных зонах ядра (использует `SLAB`-аллокатор)

- Отображение

- `kmap(page), kunmap(page)` — отобразить страницу в `highmem`, `kmap_atomic(page), kunmap_atomic(kvaddr)` — отобразить в `FIXMAP`, **не блокируется!**

Операционные системы. Часть 3. Память и планирование

All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-It LTD 1999-2020



Побольше функций тут:

<https://elixir.bootlin.com/linux/v5.4.74/source/include/linux/gfp.h#L540>:

```
/* Plain integer GFP bitmasks. Do not use this directly */
#define __GFP_DMA          0x01u
#define __GFP_HIGHMEM      0x02u
#define __GFP_DMA32         0x04u
#define __GFP_MOVABLE       0x08u
#define __GFP_RECLAIMABLE   0x10u
#define __GFP_HIGH          0x20u
#define __GFP_IO             0x40u
#define __GFP_FS             0x80u
#define __GFP_ZERO           0x100u
#define __GFP_ATOMIC          0x200u
#define __GFP_DIRECT_RECLAIM 0x400u
#define __GFP_KSMAPD_RECLAIM 0x800u
#define __GFP_WRITE           0x1000u
#define __GFP_NOWARN          0x2000u
#define __GFP_RETRY_MAYFAIL   0x4000u
#define __GFP_NOFAIL           0x8000u
#define __GFP_NORETRY          0x10000u
#define __GFP_MEMALLOC          0x20000u
#define __GFP_COMP              0x40000u
#define __GFP_NOMEMALLOC        0x80000u
#define __GFP_HARDWALL          0x100000u
#define __GFP_THISNODE          0x200000u
#define __GFP_ACCOUNT            0x400000u
#ifdef CONFIG_LOCKDEP
#define __GFP_NOLOCKDEP        0x800000u
#else
#define __GFP_NOLOCKDEP          0

```

`vzalloc` - алоцировать нулями.

`kmap` - отобразить страницу из `highmem` в память ядра (`lowmem`, которая мапится один-к-одному из-за своего расположения). В теории может быть блокирующей, то есть можем повиснуть на блокировке и нас сможет прервать процесс с более высоким приоритетом. Чтобы этого не происходило, придумали `kmap_atomic`.

106. Слаб-аллокаторы SLAB/SLUB/SLOB.

SLAB/SLUB/SLOB allocators

- Несколько реализаций, в Linux осталось 2: SLOB и SLUB
- Быстрое управление небольшими объектами
- В качестве бекенда используется alloc_page

The diagram illustrates the internal structure of the SLAB/SLUB/SLOB allocators. It starts with a `kmalloc(size, gfp_t)` call, which leads to a `kmem_cache` structure containing fields like `size`, `per_cpu`, `name`, `object_size`, `node[]`, etc. This structure points to a `kmem_cache_node` structure, which contains pointers to `slab_free`, `slab_partial`, `slab_full`, and `alien` lists. These lists point to physical memory pages. The diagram also shows a path for `another size` and `another zone` allocations, leading to additional `kmem_cache` and `kmem_cache_node` structures. Finally, the `alloc_page` function is called to handle the allocation of individual pages.

Операционные системы. Часть 3. Память и планирование
All rights reserved. Distribution is strictly prohibited unless you have written permission © Tune-It LTD 1999-2020

tuneit 29

Структуры, которые хранятся в ядре, записываются в магазины, разбитые по подстраничкам. При запросе указывается размер. Если структура такого размера существует, то попадаем в `kmem_cache`, найдем соответствующую ноду. Внутри этой ноды странички поделены на размеры, соответствующие запрашиваемому. Далее аллокатор в ноде найдет свободное место и вернет нам его.

Внутри `kmem_cache_node` есть указатель на полностью свободную страницу, на частично занятую, на полную. Аллокатор управляет записями в этих страницах.

Наиболее часто в Linux используется именно SLAB, но есть и иные аллокаторы для распределения небольших фрагментов памяти.

1. SLAB. Предназначен для работы с кешем, минимизируя, насколько возможно, количество промахов кеша.
2. SLUB (SLAB без очереди). Максимально простой с минимальным количеством команд [53].
3. SLOB (simple list of blocks — простой список блоков). Максимально компактный; предназначен для систем с ограничениями памяти [160].

107. Copy on write и pagefault в Linux.



COW — Copy On Write

- Механизм использования данных в случае их записи
 - В реальной нагрузке процесс может умереть быстрее, чем изменит страницу данных
 - Давайте при fork создавать маппинги, а не сами страницы
 - В случае обращения на чтение — просто прочитаем
 - В случае записи — создадим копию страницы для порожденного процесса (и anonymous mapping), и потом запишем
- COW порождает множество совместно используемых для чтения страниц, и копирует их при записи



do_page_fault

- Обращение к странице, которой нет в требуемом сегменте памяти
 - Minor fault — на самом деле нужные данные в памяти есть, но по разным причинам недоступны для текущего процесса
 - Major fault — frame выгружен из памяти
 - * Очищен (выгружен) для страниц кода, статических данных, ...
 - * Находится в области подкачки для анонимных страниц, ...
 - Segmentation fault — если мы обращаемся в запрещенную область
 - Kernel panic — если мы обращаемся в запрещенную область из ядра
- Может быть вызвано дефектом кода, сбоем аппаратуры,
- Основная функция `do_page_fault`

Copy-on-write: когда мы копируем области данных, можно создавать реальную копию, только когда ОС обращается к данным с целью записи, а до этого времени хранить маппинг.

minor fault - Если страница загружена в память в момент возникновения ошибки, но не помечена в MMU как загружаемая в память. Обработчик ошибок страниц в операционной системе просто должен сделать так, чтобы запись для этой страницы в блоке управления памятью указывала на страницу в памяти и было отмечено, что страница загружена в память.

major fault - Если страница не загружена в память в момент сбоя.

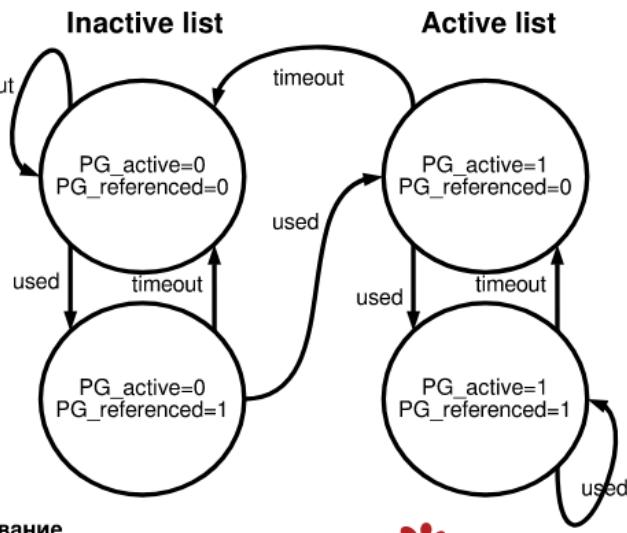
segmentation fault - скорее всего попали в неразмапленную область или в страницу с другими правами доступа или что-то еще. Другими словами, при попытке обращения к недоступным для записи участкам памяти либо при попытке изменить память запрещенным способом.

108. Замещение страниц в Linux. Kswapd.



Замещение страниц Linux. kswapd

- Для каждой зоны свой набор из двух списков page
- Кандидат на замещение — page у которой $PG_active=PG_referenced=0$
- kswapd запускается при нехватке памяти в заданной зоне
- Из списков исключаются страницы: SHM_LOCK, VM_LOCKED, ramfs



Операционные системы. Часть 3. Память и планирование

Kswapd - kernel swap daemon. Запускается при старте ядра.

Каждая зона памяти содержит два связанных списка - Active и Inactive List. Если в зоне не хватает памяти, то kswapd пытается удалять страницы, которые неактивны и не использовались в ближайшее время.

Первое обращение - нижний левый кружок, второе - правый верхний, третье - правый нижний. Обратные действия происходят по тайм-ауту.

Когда страница перестаёт быть используемой, то она обратными действиями сбрасывается в ноль, после чего подвергается пейджингу или даётся из памяти.

Страницы SHM_LOCK (shared memory), ramfs, VM_LOCKED исключаются из алгоритма.

Дополнительно (у Клименкова нет):

Каждый раз, когда таймер истекает, демон подкачки проверяет, не становится ли слишком мало свободных страниц в системе. Он использует две переменные, free_pages_high и free_pages_low, чтобы решить, следует ли освобождать некоторые страницы. Пока количество свободных страниц в системе остается выше free_pages_high, демон подкачки ядра ничего не делает; он снова спит, пока не истечет его таймер.

$$(nr_free_pages + nr_slab_free_pages < free_pages_high)$$

Для целей этой проверки демон подкачки ядра учитывает количество страниц, записанных в данный момент в файл подкачки. Он ведет их подсчет в `nr_async_pages`; это значение увеличивается каждый раз, когда страница ставится в очередь, ожидая записи в файл подкачки, и уменьшается, когда запись на устройство подкачки завершена. `free_pages_low` и `free_pages_high` устанавливаются во время запуска системы и связаны с количеством физических страниц в системе. Если количество свободных страниц в системе упало ниже `free_pages_high` или, что еще хуже, `free_pages_low`, демон подкачки ядра попытается уменьшить количество физических страниц, используемых системой, тремя способами:

- Уменьшение размера буферного и страничного кеша,
- Свопинг разделяемых страниц
- Свопинг или удаление страниц

Если падает ниже `free_pages_low`, то начинает свопиться в 2 раза больше страниц, `kswapd` начинает просыпаться в 2 раза чаще.