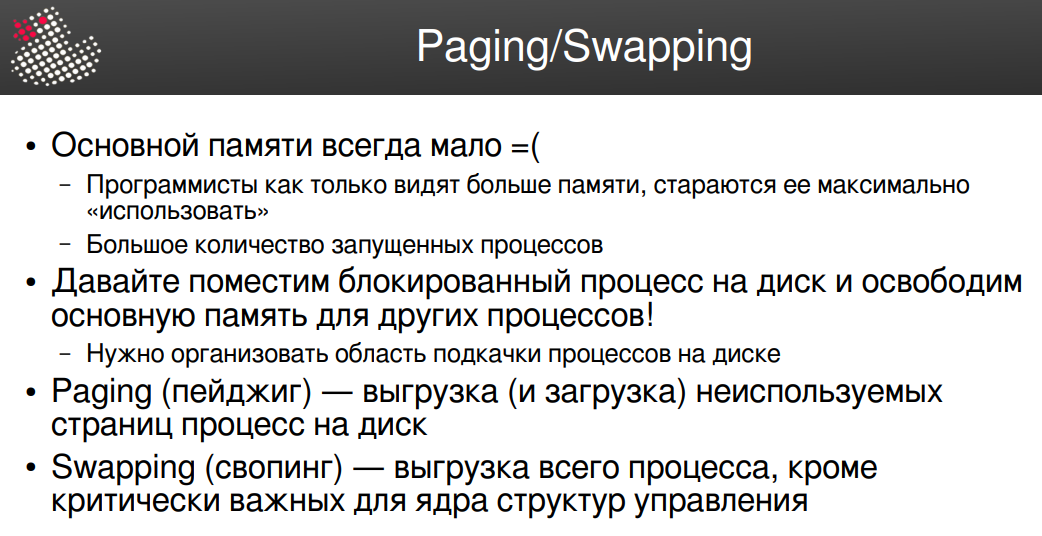
1. Paging и Swapping. Модель процесса с семью состояниями. 分页和交换。 七态过程模型。



分页/交换

主内存总是不够用 =(

- 程序员一看到内存增多，就会尽可能地 "使用 "它

大量进程正在运行

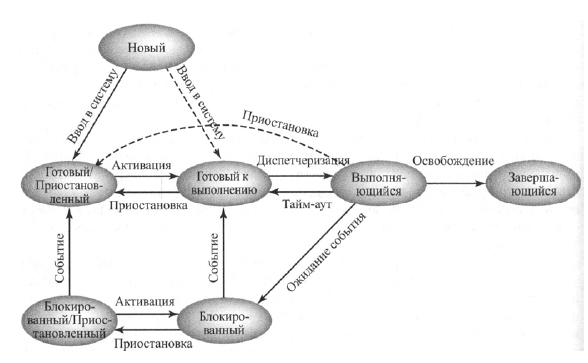
让我们把阻塞的进程放到磁盘上，为其他进程释放主内存！

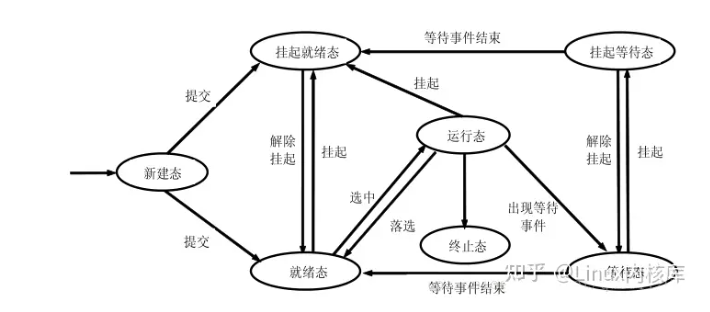
- 我们需要为磁盘上的进程组织一个交换区

分页 -- 将未使用的进程页卸载（和加载）到磁盘上。

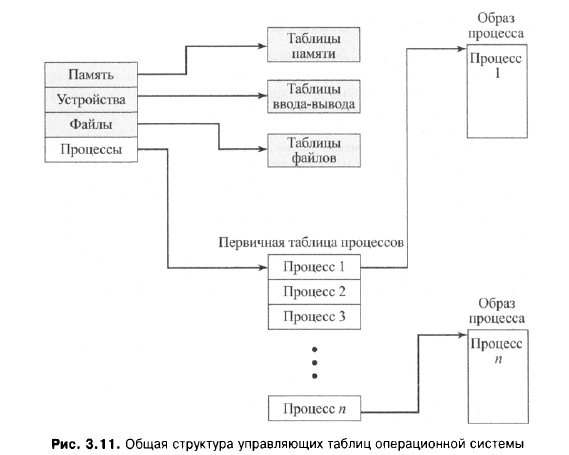
交换--卸载整个进程，内核关键控制结构除外。

Модель процесса с 7 состояниями         七态过程模型





1. Управляющие таблицы процесса. Образ процесса. 过程控制表。 过程图像。



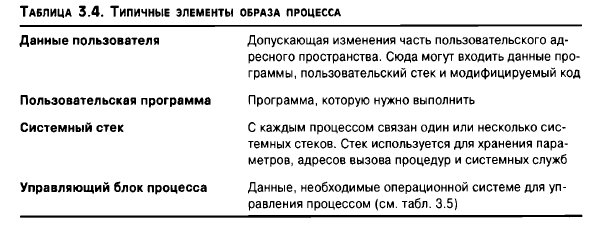


表 3.4. 流程图像的典型要素

用户数据

用户地址空间的可修改部分。这可能包括程序数据、用户堆栈和可修改代码

用户程序

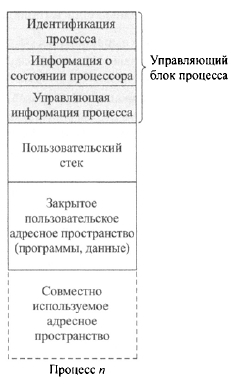
要执行的程序

系统堆栈

每个进程都有一个或多个系统栈。堆栈用于存储参数、过程调用地址和系统服务

进程控制块

操作系统控制进程所需的数据（见表 3.5）。



鉴定 识别过程

有关 处理器状态 控制 处理单元

控制 过程控制信息

用户 堆栈

关闭 用户 地址空间 (程序、数据）

共享 共享的 地址空间 地址空间

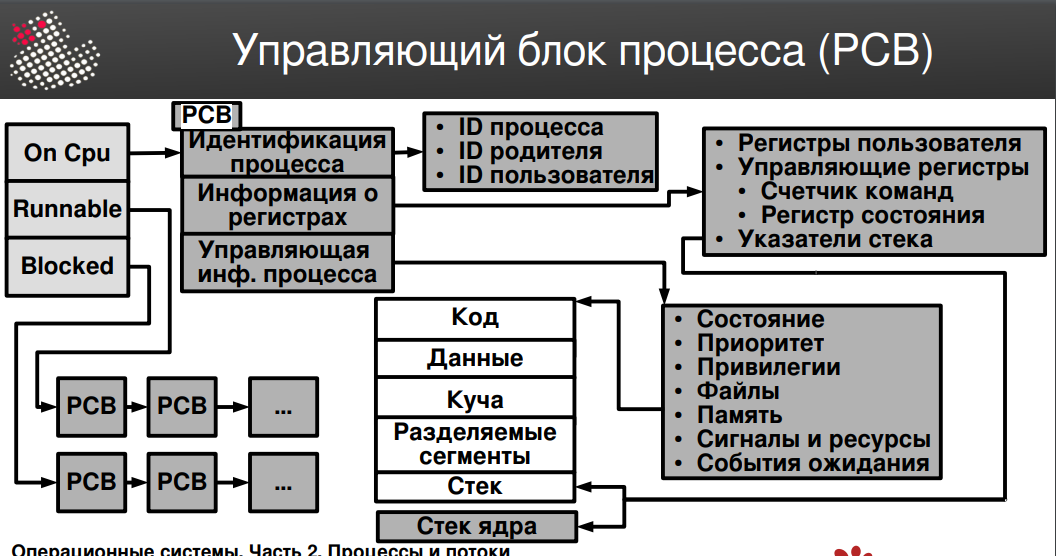
进程 n

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

过程控制块（PCB），PCB组成。

进程控制块的作用

进程控制块是操作系统中最重要的数据结构。进程控制块包含操作系统需要的关于每个进程的所有信息。操作系统中的几乎所有模块（包括与调度、资源分配、中断处理以及监控和分析相关的模块）都会读取和/或修改这些块中的信息。操作系统的状态可以说是由一组进程控制块决定的。



它们包含有关进程、其 ID、寄存器状态、代码、数据等的所有信息。В них содержится вся информация о процессе, его ID, состояния регистров, код, данные и т.д.

1. Функции ОС, связанные с процессами. Создание процесса, переключение процессов.  与进程相关的操作系统功能。 进程创建，进程切换。

创建流程

•为进程分配唯一标识符

•为进程分配内存

\*初始化RSV

•将进程放入内核队列

\*创建I/O流

•创建其他控制数据结构

切换过程

•该进程可以在用户和内核模式下运行

\*使用中断机制:

-外部中断(IO)

-陷阱-错误或异常处理

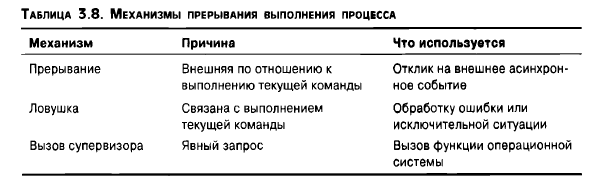
-调用操作系统

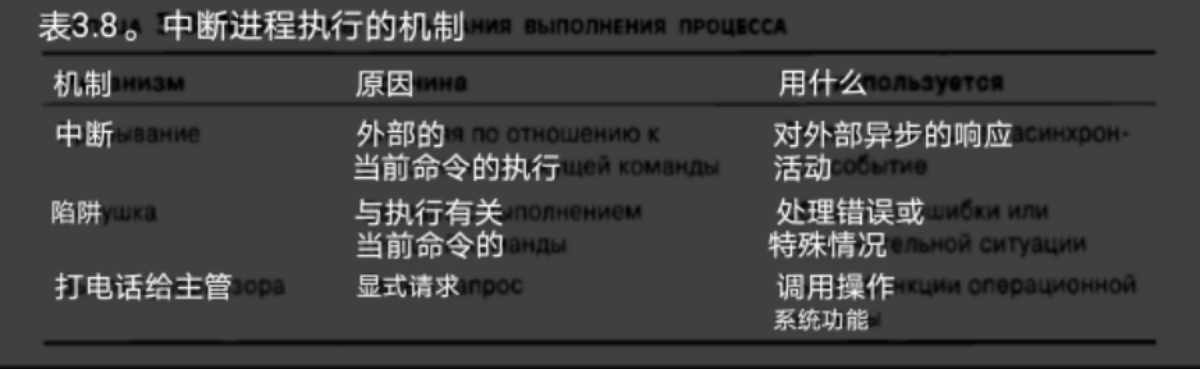
•从用户切换到内核模式的情况:

-定时器中断

-I/O中断

-页面故障-缺失内存块



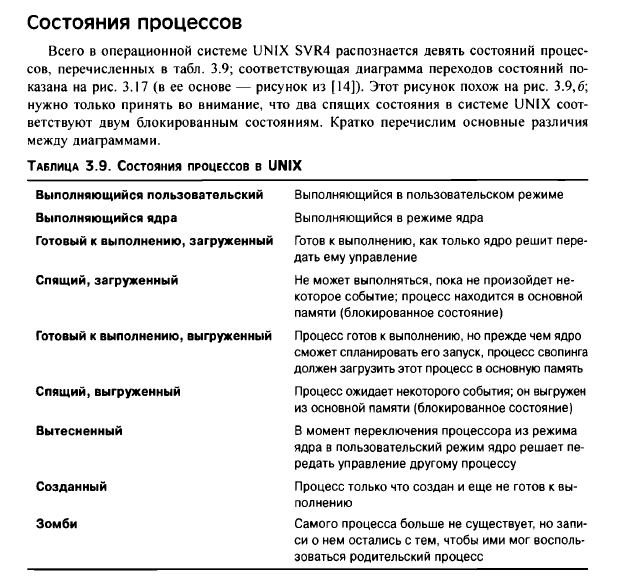


- 定时器中断。操作系统确定当前进程运行的最大允许时间，称为时间片。时间片是指进程可以不间断运行的最长时间。在这种情况下，必须将进程切换到就绪状态，并将控制权转移到另一个进程。

- I/O 中断。操作系统会判断发生了什么情况，如果是一个或多个进程正在等待的事件，则操作系统会将所有相关的阻塞进程置于就绪状态（分别将阻塞/暂停进程置于就绪/暂停状态）。然后，操作系统必须决定是恢复执行当前进程，还是将控制权转移到优先级更高的就绪进程。

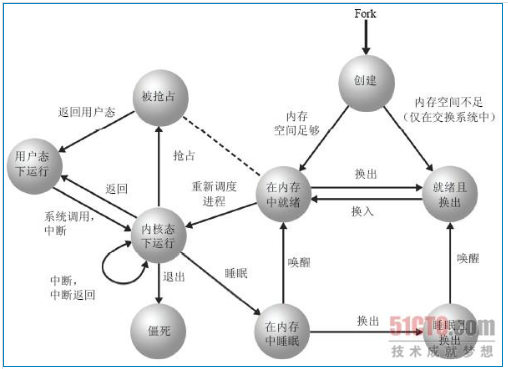
- 内存块丢失错误。假设处理器需要访问主内存中当前丢失的虚拟内存字。在这种情况下，操作系统必须将包含所寻址字的内存块（页或段）加载到主内存中。在请求加载块后，操作系统会立即将控制权转移给另一个进程，而需要将块加载到主内存中才能继续执行的进程会进入阻塞状态。当所需数据块加载完毕后，进程进入就绪状态。

28. Процессы в ОС UNIX SVR4. Диаграмма состояний, основные структуры.           UNIX OS SVR4 中的进程。 状态图，基本结构。

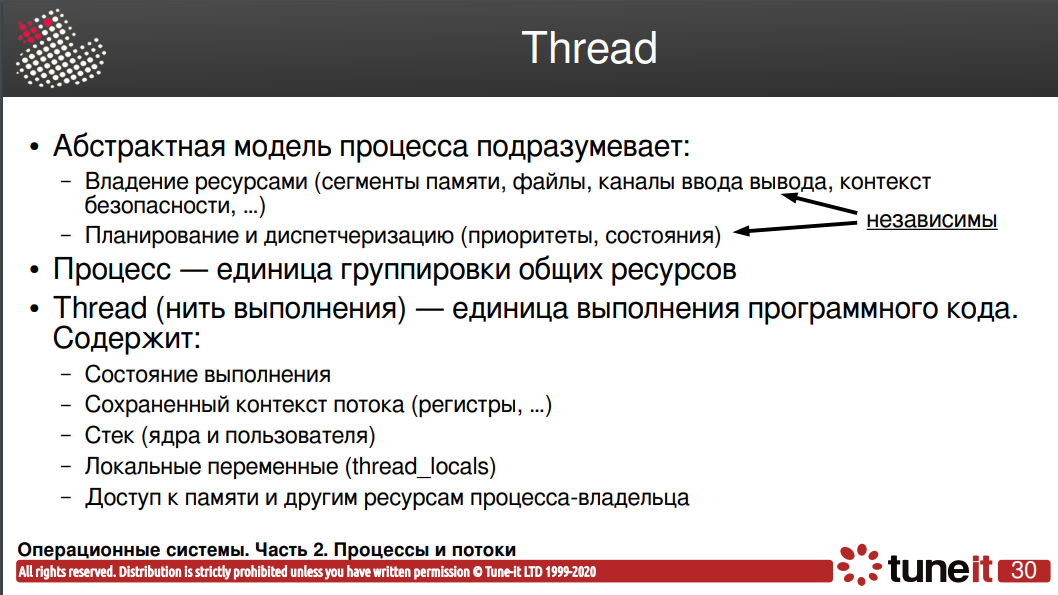


进程状态 UNIX SVR4 操作系统总共可识别表 3.9 所列的九种进程状态；相应的状态转换图如图 3.17 所示（基于 [14] 中的图表）。3.17 （根据 [14] 中的图表绘制）。该图与图 3.9.6 相似，只是图 3.9.6 的状态转换图与图 3.9.6 的状态转换图不同。3.9.6 相似；只需考虑到 UNIX 系统中的两个睡眠状态对应两个阻塞状态。让我们简要列出两图的主要区别。





29. Понятие потока выполнения, связь потока и процесса. Преимущества потоков. 线程执行的概念，线程与进程的连接。 线程的好处。



- 抽象流程模型意味着

- 资源（内存段、文件、输出输入通道、安全环境等）的所有权

-调度和分派（优先级、状态）

- 进程是共享资源的分组单位

-线程（执行线程）--软件代码的执行单位。

包含：

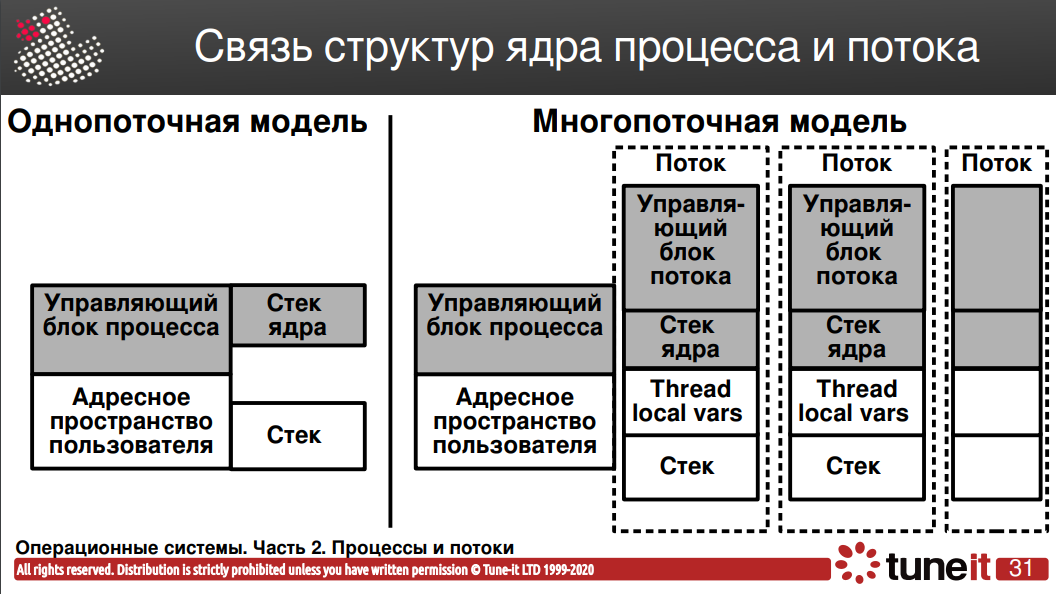
- 执行状态

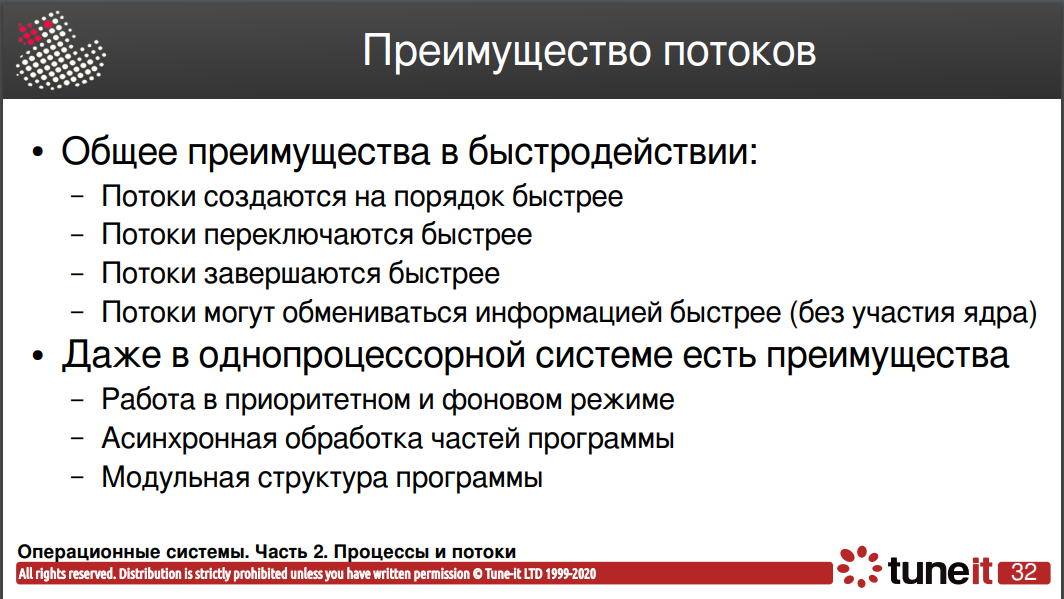
- 线程存储的上下文（寄存器、...）

- 堆栈（内核和用户）

- 局部变量（thread\_locals）

- 访问拥有进程的内存和其他资源





流的优势

-快速性能方面的一般优势

- 流的创建速度快一个数量级

- 线程切换更快

- 线程终止速度更快

- 线程可以更快地交换信息（无需内核）。

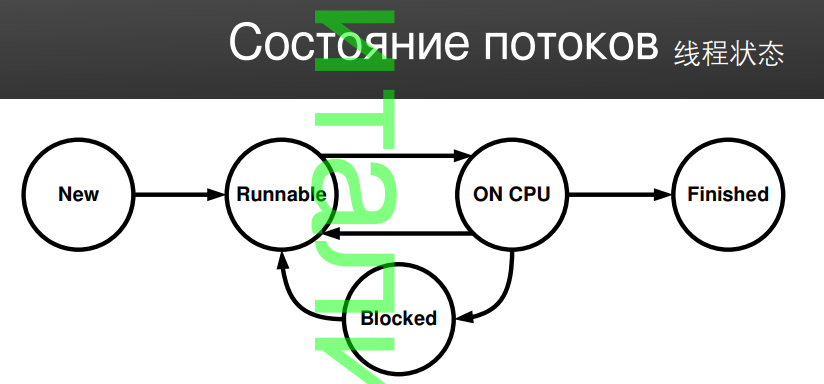
-即使在单处理器系统中也有优势。

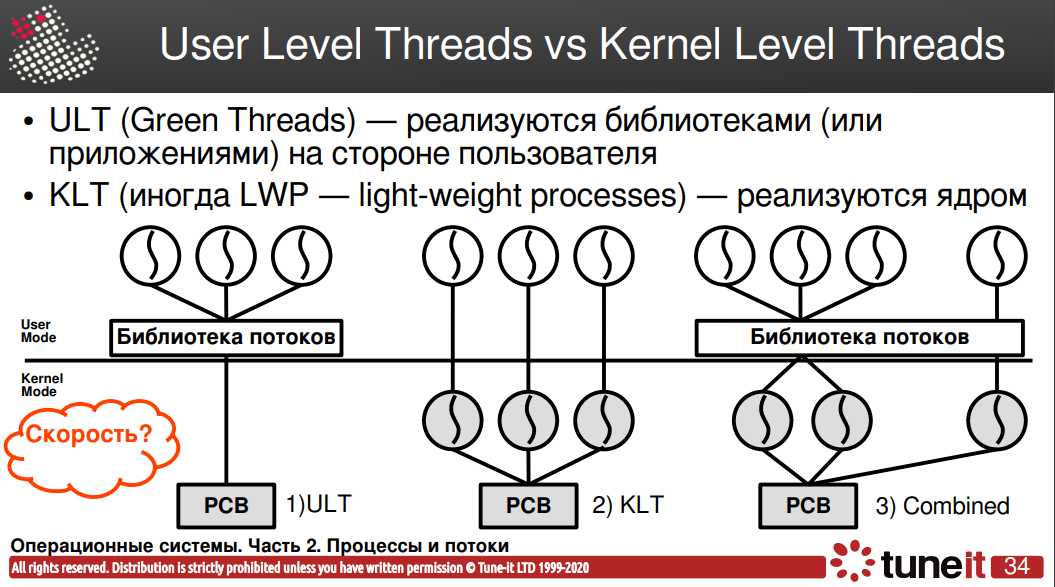
- 在优先和后台模式下工作

- 程序各部分的异步处理

- 模块化程序结构流的优势

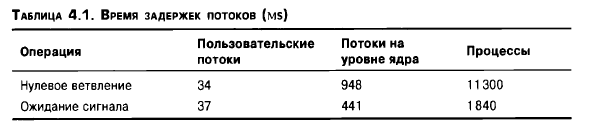
1. Состояния потока, User Level Threads vs Kernel Level Threads      线程状态、用户级线程与内核级线程

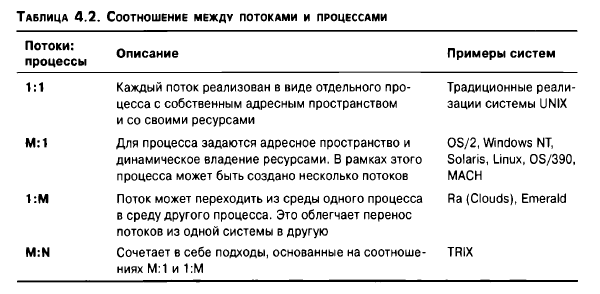




- ULT（绿色线程） - 由用户侧的库（或应用程序）实现

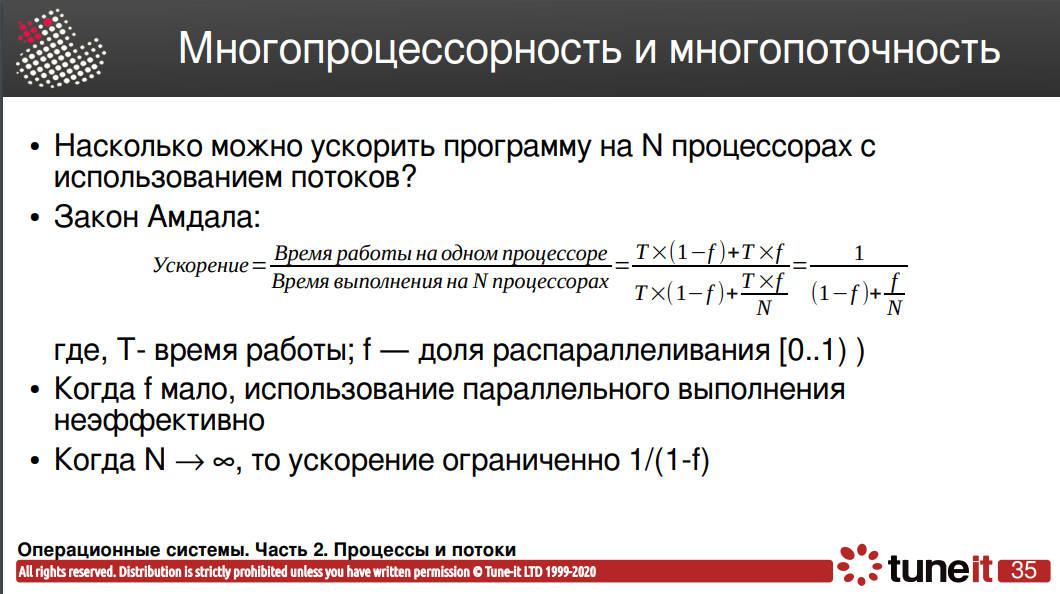
- KLT（有时为 LWP--轻量级进程）--由内核实现

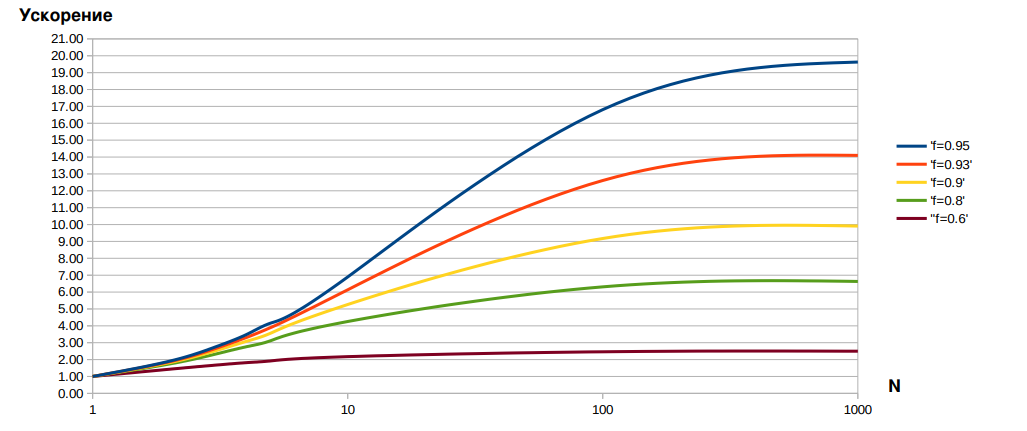




|  |  |
| --- | --- |
| 每个线程都作为一个独立的进程来实现，拥有自己的地址空间和资源 | 传统的 UNIX 系统实现 |
| 为进程定义地址空间和动态资源所有权。该进程内可创建多个线程 | OS/2、Windows NT、 Solaris、Linux、OS/390、 MACH |
| 线程可以从一个进程环境移动到另一个进程环境。这使得将线程从一个系统移动到另一个系统变得非常容易 | Ra (Clouds), Emerald |
| 结合 M:1 和 1:M 比率方法 | TRIX |

1. Многопроцессорность и многопоточность. Закон Амдала.         多处理和多线程。 阿姆达尔定律。





## Механизм параллельных вычислений, функции ОС. 并行计算机制，OS功能。

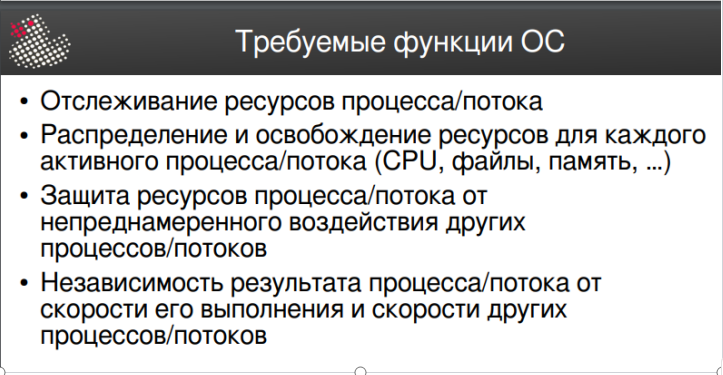


“并行计算”的机制

单处理器系统—候补进程

多处理器—交替和重叠

共享资源的竞争



所需的操作系统功能

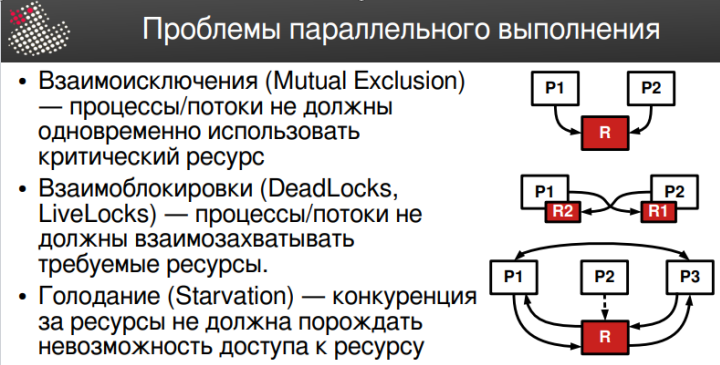
-跟踪进程/线程资源

-分配和释放每个活动进程/线程的资源（CPU、文件、内存...）

-保护进程/线程资源不受其他进程/线程的意外影响

-进程/线程的结果与其执行速度和其他进程/线程的速度无关

1. Проблемы параллельного выполнения: взаимоисключения, взаимоблокировки, голодание. Требования к взаимным исключениям. Уровни взаимодействия процессов и потоков. 并行执行问题：互斥、互阻、饥饿。对互斥的要求 进程与线程之间的交互层次。



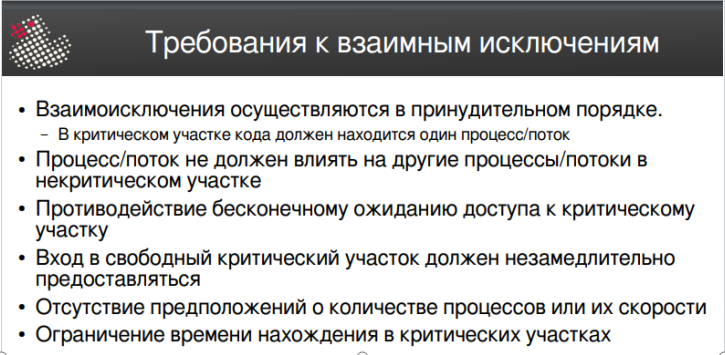
并行完成的问题

-相互排除（Mutual Exclusion）

- 进程/线程不得同时使用关键资源

-死锁（DeadLocks）、活锁（LiveLocks） -- 进程/线程不应截获所需的资源。

- 饥饿（Starvation） -- 资源竞争不应导致无法访问资源。



相互排除的要求

- 强制执行互斥。

- 关键代码段中必须有一个进程/线程

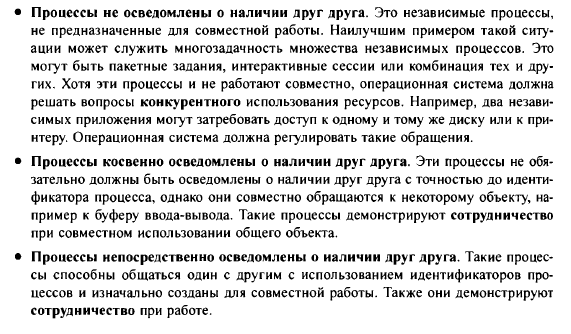
-进程/线程不得影响非关键代码段中的其他进程/线程

- 防止无休止地等待访问关键代码段

- 应能立即访问未占用的关键区域

- 不假定流程的数量或速度

- 限制花费在关键区域的时间



-进程彼此不知道对方的存在。它们是独立的进程，并不是为了一起工作而设计的。这种情况最典型的例子就是多任务处理中的多个独立进程。这些进程可能是批处理作业、交互会话或两者的组合。虽然这些进程并不协同工作，但操作系统必须处理竞争性资源利用问题。例如，两个独立的应用程序可能会请求访问同一个磁盘或打印机。操作系统必须对此类请求进行管理。

-进程间接了解彼此的可用性。这些进程不一定要知道对方是否存在，但它们可以共享对一个对象（如 I/O 缓冲区）的访问。这类进程在共享一个共同对象时表现出合作精神。

-进程直接知道对方的存在。这些进程能够使用进程标识符相互通信，并且从一开始就被设计为一起工作。它们在协同工作时也表现出合作精神。

1. Примитивы синхронизации ОС. Предназначение примитивов синхронизации               操作系统同步的原语。 同步原语的用途



基本基元 - Semaphores - 捕捉和释放多个资源或单个资源（二进制 Semaphores）

Mutex--由单个进程/线程锁定和释放资源。

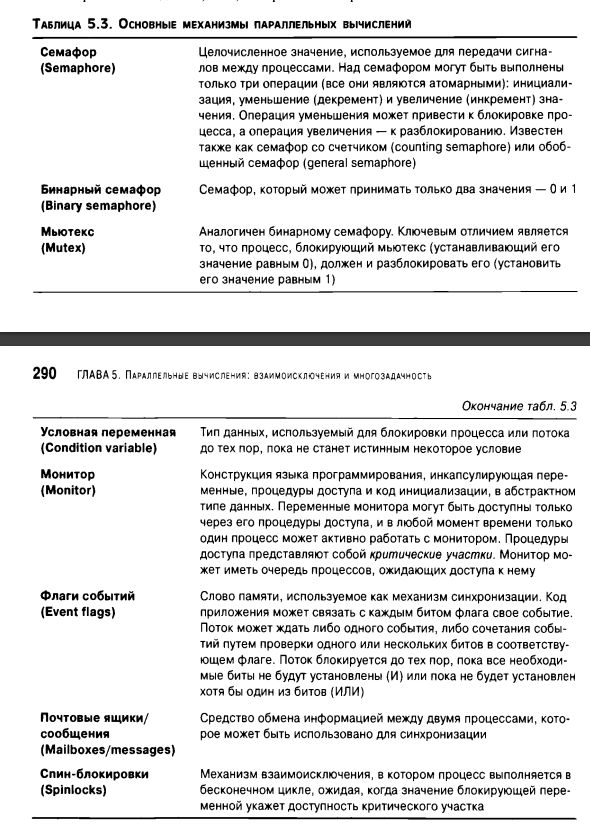
-条件变量 --锁定，直到满足条件。

读写锁（rw-locks）--独立的读写锁。

-监控器（Monitors）-- 隐藏低级同步原语的编程语言结构。

-事件标志 -- 将执行继续条件与一个或多个标志（阻塞变量的位）联系起来。

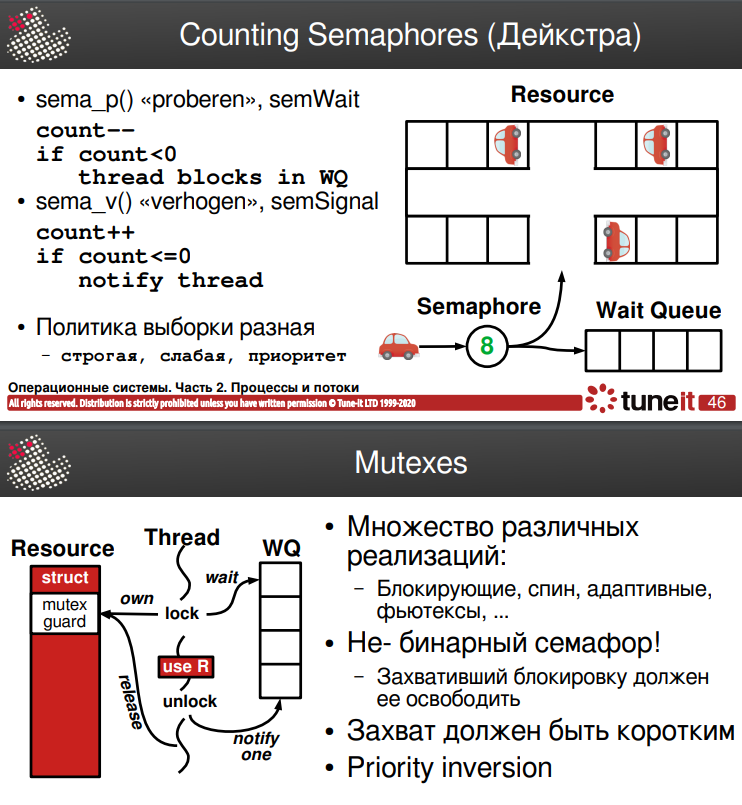
-消息传递 -- 消息传递



1. Примитивы синхронизации ОС. Семафоры и мьютексы. Бинарный семафор           操作系统同步的原语。 信号量和互斥量。 二进制信号量

Семафоры (Semaphore) — захват и освобождение множественного ресурса – или одного (бинарные семафоры)    信号量 - 多个资源的占用和释放 - 或一个（二进制信号量）

Мьютексы (Mutex) — блокировка и освобождение ресурса единственным процессом/потоком     互斥锁 - 单个进程/线程阻塞和释放资源

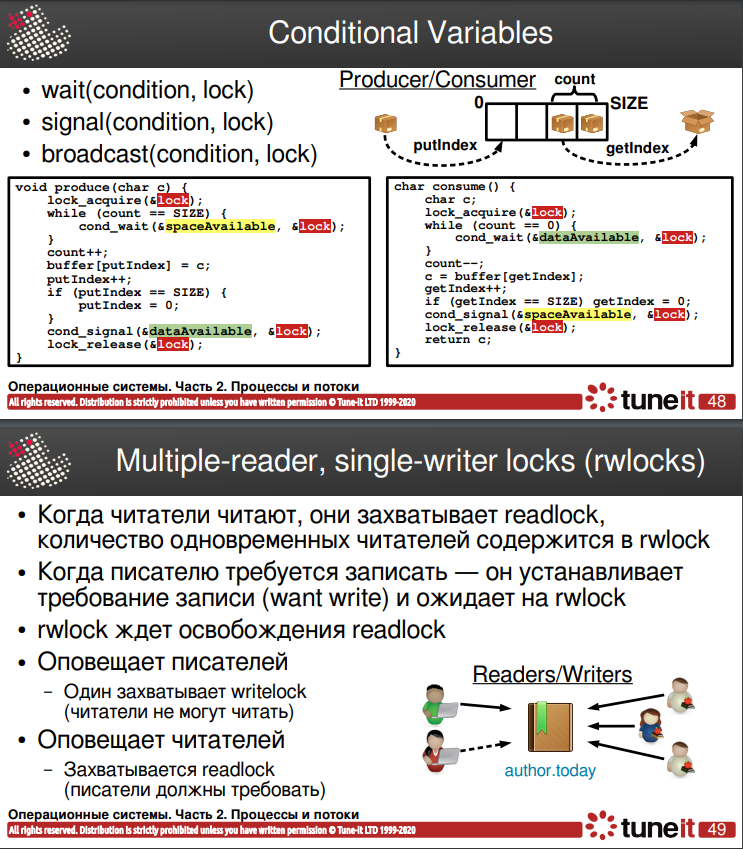


1. Примитивы синхронизации ОС. Условные переменные, rwlocks. 操作系统同步的原语。 条件变量，rwlocks。

Условные переменные (Conditional Variable) — Блокировка до выполнения какого-либо условия         条件变量 - 阻塞直到满足条件

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

读/写锁（rw-lock）——单独的读和写锁



多读单写锁 (rwlocks)

-当读取器读取时，它们会捕获一个读取锁，同时读取器的数量包含在 rwlock 中。

- 当写入者需要写入时--它会设置写入需求（想要写入）并等待 rwlock

-rwlock 等待读锁释放

-通知写入器

- 一个写入锁被夺取

(读者无法读取）

-通知读者

- 一个捕获读锁

(写入器必须要求）

1. Примитивы синхронизации ОС. Мониторы, флаги событий, передача сообщений.操作系统同步原语。监控器、事件标志、消息传递。

Мониторы (Monitor) — конструкции языков программирования, которые скрывают низкоуровневые примитивы синхронизации         监视器 - 隐藏低级同步原语的编程语言结构

Флаги событий (Event Flags) — связывание условий продолжения выполнения с одним или несколькими флагами (битами блокирующей переменной)            事件标志 - 使用一个或多个标志（阻塞变量的位）继续执行的绑定条件

Почтовые ящики (Message Passing) — передача сообщений           邮箱（消息传递） - 传递消息

1. Примитивы синхронизации ОС. Неблокирующие примитивы синхронизации и неблокирующие структуры данных.                操作系统同步的原语。 非阻塞同步原语和非阻塞数据结构。

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

非阻塞算法的优点是在处理器数量方面具有更好的可扩展性。 此外，如果操作系统用后台任务中断其中一个线程，其余线程至少会在不空闲的情况下完成它们的工作。 在最大程度上，他们将接管出色的工作。

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

现在流行使用非阻塞同步原语和非阻塞结构 - 无等待（N 步）、无锁（一些 N 步，其他重试）、无阻塞

Без препятствий (англ. obstruction-free)          无障碍

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

最弱的保证。 如果一个流没有遇到来自其他流的障碍，它就会取得进展。 如果一个线程在任何时候启动（假设所有阻塞线程的执行都被挂起）以确定的步数完成其工作，则该算法可以无障碍地工作。 使用互斥锁的同步甚至不能满足这个要求：如果一个线程通过捕获一个互斥锁而停止，那么其他需要这个互斥锁的线程将处于空闲状态。

Без блокировок ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) lock-free)               无锁

Для алгоритмов без блокировок гарантируется системный прогресс по крайней мере одного потока. Например, поток, выполняющий операцию «[сравнение с обменом](https://ru.wikipedia.org/wiki/%D0%A1%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81_%D0%BE%D0%B1%D0%BC%D0%B5%D0%BD%D0%BE%D0%BC)» 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.

对于无阻塞算法，至少保证一个线程的系统进度。 例如，一个线程在循环中执行“比较交换”操作，理论上可以无限期运行，但它的每一次迭代都意味着其他某个线程取得了进展，即整个系统都在取得进展。

Без ожиданий ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) wait-free)         免等待

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

最严格的进步保障。 如果每个操作在一定数量的步骤中执行，该算法无需等待即可工作，独立于其他线程

Примеры:        例子：

Атомарные операции и примитивы синхронизации       原子操作和同步原语

Неблокирующие структуры данных — списки, деревья, очереди, … over 9000.      非阻塞数据结构 - 列表、树、队列……超过 9000 个。

1. Управление памятью, основные определения и требования к организации.           内存管理、组织的基本定义和要求。



**Перемещение          移动**

 В многозадачной системе доступная основная память в общем случае разделяется среди множества процессов. Обычно программист не знает заранее, какие программы будут резидентно находиться в основной памяти во время работы разрабатываемой им программы. Кроме того, для максимизации загрузки процессора желательно иметь большой пул процессов, готовых к выполнению, для чего требуется возможность загрузки и выгрузки активных процессов из основной памяти. Требование, чтобы выгруженная из памяти программа была вновь загружена в то же самое место, где находилась и ранее, было бы слишком сильным ограничением. Крайне желательно, чтобы программа могла быть перемещена (relocate) в другую область памяти. Таким образом, заранее неизвестно, где именно будет размещена программа, а кроме того, программа может быть перемещена из одной области памяти в другую при свопинге. Эти обстоятельства обусловливают наличие определенных технических требований к адресации, проиллюстрированных на рис. 7.1.

在多任务系统中，可用的主内存通常在多个进程之间共享。通常，程序员在他正在开发的程序正在运行时，事先并不知道哪些程序将驻留在主存储器中。此外，为了最大限度地提高处理器利用率，最好有一个大的准备运行的进程池，这需要能够从主内存加载和卸载活动进程。从内存中卸载的程序重新加载到它之前所在的位置的要求太强了。非常希望程序可以重新定位到不同的内存区域。因此，事先并不知道程序的确切位置，此外，程序可以在交换期间从一个内存区域移动到另一个内存区域。这些情况导致存在某些寻址技术要求，如图 1 所示。 7.1。



На рисунке представлен образ процесса. Для простоты предположим, что образ процесса занимает одну непрерывную область основной памяти. Очевидно, что операционной системе необходимо знать местоположение управляющей информации процесса и стека выполнения, а также точки входа для начала выполнения процесса. Поскольку управлением памятью занимается операционная система и она же размещает процесс в основной памяти, соответствующие адреса она получает автоматически. Однако, помимо получения операционной системой указанной информации, процесс должен иметь возможность обращаться к памяти в самой программе. Так, команды ветвления содержат адреса, указывающие на команды, которые должны быть выполнены после них; команды обращения к данным - адреса

该图显示了该过程的图像。为简单起见，我们假设过程映像占用主存的一个连续区域。显然，操作系统需要知道进程控制信息和执行栈的位置，以及启动进程的入口点。由于操作系统参与了内存管理，而且它也在主存中分配进程，所以它会自动接收相应的地址。但是，进程除了接收操作系统指定的信息外，还必须能够访问程序本身的内存。例如，分支命令包含指示要在它们之后执行的命令的地址；数据访问命令 - 地址

байтов или слов, с которыми они работают. Так или иначе, но процессор и программное обеспечение операционной системы должны быть способны перевести ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти.

它们使用的字节或单词。 一种或另一种方式，处理器和操作系统软件必须能够将程序代码中的链接转换为与程序在主存储器中的当前位置相对应的真实物理地址。

**Защита      保护**

 Каждый процесс должен быть защищен от нежелательного воздействия других процессов, случайного или преднамеренного. Следовательно, код других процессов не должен иметь возможности без разрешения обращаться к памяти данного процесса для чтения или записи. Однако удовлетворение требованию перемещаемости усложняет задачу защиты.

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

必须保护每个过程免受其他过程的意外或故意影响。因此，其他进程的代码应该不能在未经许可的情况下访问该进程的内存进行读写。然而，满足可移植性的要求使安全任务复杂化。

由于程序在主存中的位置是不可预测的，因此无法在编译时检查绝对地址。此外，在大多数编程语言中，可以在运行时动态计算地址（例如，计算数组元素的地址或数据结构中指向字段的指针）。因此，在程序运行时，需要检查进程产生的所有内存访问，以确保它们都只是分配给该进程的内存。幸运的是，正如您稍后将看到的，移动支持机制也提供了保护支持。通常，用户进程不能访问操作系统的任何部分——既不能访问代码也不能访问数据。一个进程中的代码不能执行目标代码在另一个进程中的分支命令。除非采取特殊预防措施，否则来自一个进程的代码无法访问来自另一个进程的数据。处理器必须能够中断此类指令的执行。请注意，内存保护要求必须在处理器（硬件）级别满足，而不是在操作系统（软件）级别，因为操作系统无法预测程序将进行的所有内存访问。即使有可能，扫描每个程序以查找建议的安全漏洞在 CPU 使用方面也太浪费了。因此，相应的硬件能力是确定程序运行时访问内存（数据或代码）的可允许性的唯一手段

**Совместное использование          分享**

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

任何保护机制都必须足够灵活，以允许多个进程访问主存的同一区域。例如，如果多个进程正在执行相同的机器代码，那么让每个进程使用该代码的相同副本而不是创建自己的副本将是有益的。在任务上协作的进程可能需要共享相同的数据结构。因此，内存管理系统应该提供对共享内存区域的受控访问，而不会以任何方式损害内存保护。正如我们稍后将看到的，移动支持机制也提供了对内存共享的支持。

**Логическая организация        逻辑组织**

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

几乎总是，计算机系统中的主存储器被组织为线性（一维）地址空间，由一系列字节或字组成。辅助存储器在其物理级别以类似的方式组织。虽然这种组织反映了所使用的特定硬件，但它与通常编写程序的方式不匹配。大多数程序都是按模块组织的，其中一些是不可变的（只读、只执行），而另一些则包含可以更改的数据。如果计算机的操作系统和硬件能够有效地处理模块所代表的用户程序和数据，则有许多优点。

1. Модули могут быть созданы и скомпилированы независимо один от другого, при этом все ссылки из одного модуля во второй разрешаются системой во время работы программы.       模块可以相互独立地创建和编译，一个模块对第二个模块的所有引用在程序运行时由系统解析。

 2. Разные модули могут получить разные степени защиты (только для чтения, только для выполнения) за счет весьма умеренных накладных расходов.      2. 不同的模块可以以非常适中的开销实现不同程度的保护（只读、只执行）。

3. Возможно применение механизма, обеспечивающего совместное использование модулей разными процессами.    3.可以使用确保不同进程共享模块的机制。

Основное достоинство обеспечения совместного использования на уровне модулей заключается в том, что они соответствуют взгляду программиста на задачу и, следовательно, ему проще определить, требуется ли совместное использование того или иного модуля. Инструментом, наилучшим образом удовлетворяющим данным требованиям, является сегментация, которая будет рассмотрена в данной главе среди прочих методов управления памятью.

提供模块级共享的主要好处是它与程序员的任务视图保持一致，因此使他更容易确定是否需要共享模块。 最适合这些要求的工具是分段，本章将介绍它以及其他内存管理技术。

**Физическая организация      物理组织**

Как указывалось в разделе 1.5, память компьютера разделяется как минимум на два уровня: основная и вторичная. Основная память обеспечивает быстрый доступ по относительно высокой цене; кроме того, она энергозависима, т.е. не обеспечивает долговременное хранение. Вторичная память медленнее и дешевле основной и обычно энергонезависима. Следовательно, вторичная память большой емкости может служить для долговременного хранения программ и данных, а основная память меньшей емкости - для хранения программ и данных, использующихся в текущий момент. В такой двухуровневой структуре основной заботой системы становится организация потоков информации между основной и вторичной памятью. Ответственность за эти потоки может быть возложена и на отдельного программиста, но это непрактично и нежелательно по следующим причинам.

**如 1.5 节所述，计算机内存至少分为两个级别：初级和次级。主存储器以相对较高的成本提供快速访问；此外，它是易变的，即不提供长期存储。辅助存储器比主存储器更慢且更便宜，并且通常是非易失性的。因此，较大的辅助存储器可用于长期存储程序和数据，而较小的主存储器可用于存储当前正在使用的程序和数据。在这样的两级结构中，系统的主要关注点是主存储器和辅助存储器之间信息流的组织。这些线程的责任可以分配给单个程序员，但由于以下原因，这是不切实际和不可取的。**

1. Основной памяти может быть недостаточно для программы и ее данных. В этом случае программист вынужден прибегнуть к практике, известной как структуры с перекрытием - оверлеи (overlay), когда программа и данные организованы таким образом, что различные модули могут быть назначены одной и той же области памяти; основная программа при этом ответственна за перезагрузку модулей при необходимости. Даже при помощи соответствующего инструментария компиляции оверлеев разработка таких программ приводит к дополнительным затратам времени программиста.

 1. 主存可能不足以容纳程序及其数据。在这种情况下，程序员不得不求助于一种称为覆盖的做法，其中程序和数据的组织方式使得不同的模块可以分配到相同的内存区域。主程序负责在必要时重新加载模块。即使使用适当的覆盖编译工具包，开发此类程序也会导致额外的程序员时间支出。

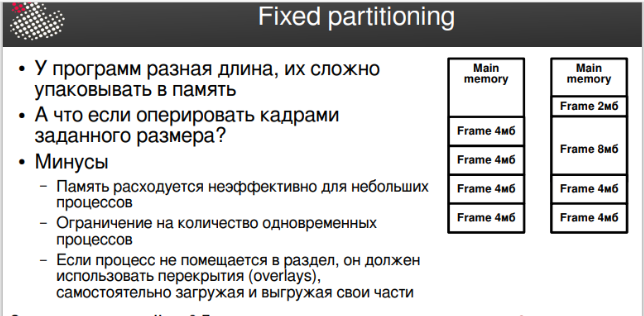
 2. Во многозадачной среде программист при разработке программы не знает, какой объем памяти будет доступен программе и где эта память будет располагаться.

Таким образом, очевидно, что задача перемещения информации между двумя уровнями памяти должна возлагаться на операционную систему. Эта задача является сущностью управления памятью.

2. 在多任务环境中，程序员在开发程序时，不知道该程序可以使用多少内存以及该内存将位于何处。

因此，很明显，在两级内存之间移动信息的任务应该分配给操作系统。 这个任务是内存管理的本质。

1. Фиксированное и динамическое размещение программ в памяти.                程序在内存中的固定和动态放置。



固定分区

-程序长短不一，很难将它们打包到内存中

-如果我们对给定大小的帧进行操作呢？

-缺点

-小进程的内存使用效率低

-同时运行的进程数量有限

-如果一个进程无法装入一个分区，它就必须使用覆盖来独立加载和卸载其部分。



动态分区

-处理不同大小的区块，操作系统将其构建为可用空间

-内存

- 内存碎片

- 内存打包的必要性

- 善始善终，你懂的....

- 复杂而缓慢的内存放置算法。

-适合加载驱动程序。

- 这就是为什么我们在添加驱动程序时需要重启。

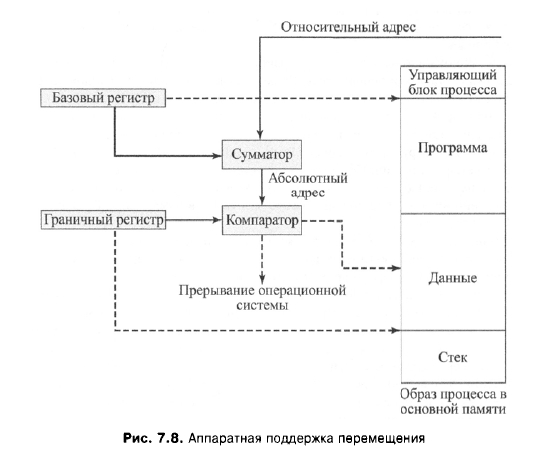
- 一个合理的折中方案是 Buddy 系统 - 参见 Stallings。

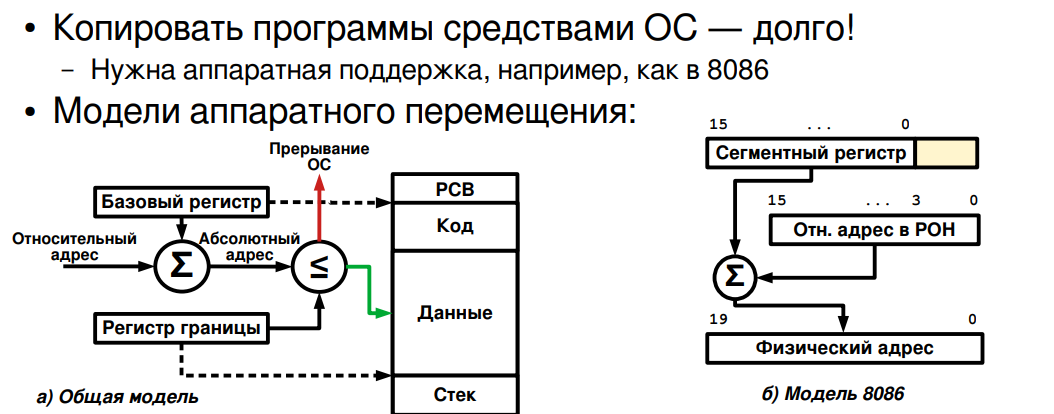
固定分区的关键是操作系统将内存分为几个大小相同的帧,便于管理.,如果程序大于帧将无法放入,小于帧又会出现空间剩余.

动态分区   需要先打包空闲的内存碎片,按顺序排列,系统开始工作很快,到了打包的时候性能就会急剧下降,对比固定分区,我们可以快速的查找内存的占用情况,利用算法来合理的分配内存中的进程.

41. Модели аппаратного перемещения программ.

程序的硬件运动模型。





基址寄存器=指示程序的开始

边界寄存器- 表示程序的结束.

在用户程序中,相对地址被编码为偏移量.相对地址会进入加法器,然后添加到基址寄存器中,会产生一个绝对内存地址. 边界寄存器是用来检查绝对地址是否超出边界.堆栈通常是由特殊命令控制,如果数据适合堆栈,它们就会重叠,这是不对的.所以边界寄存器可以防止程序越界.

42. Простой страничный поход и простая сегментная организация. 简单的页面活动和简单的段组织。



分页是将内存分为大小相同且足够小的帧,一般帧的大小为2的倍数.

如果一个进程占用1.5帧那么它将占用系统的主内存为两帧.

除了分页,还需要维护页表.给被进程占用的帧编号,每个进程都应该有这样的表.操作系统有必要在每次新进程启动时,为自己创建负载表.可以维护所有页面的负载表.

另外一种方式是 简单分段   ,是使用分段表来实现,该表通常位于寄存器和RAM中.该表描述了段长度和一些基地址.,在地址空间中,程序的排列方式是每个位置在表中创建一个地址.结果在这种情况下没有内部分段,因为该分段的空间和我们程序一样多.并且外部分割严重减少,不需要打包进程.   简单分段的优点不需要移动进程,只需要在段表中指出他们的特征,把该进程占用的内存与分配的一样多.

1. Виртуальная память основные определения и принципы организации аппаратуры и управляющих программ.      虚拟内存基本定义和设备和控制程序的组织原则。

虚拟地址通过各种算法转为物理地址.

物理内存由处理器作为具有地址的顺序内存块寻址.

虚拟地址是进程内的逻辑地址.

虚拟地址空间是进程内的空间

主存储器和页存储器都具有相同的寻址方法,所以操作系统可以从主内存和辅助内存中创建空间. 虚拟地址被翻译成主存地址, 使用寻址方法和进程的虚拟地址,将主内存扩展为次内存.

内存的大小受寻址方案的限制,而不是物理地址空间的大小.

**Виртуальное адресное пространство** — область для одного процесса с виртуальными адресами

**Виртуальная память** — схема расположения процессов в памяти, в которой:

– Вторичная память адресуется так-же как и основная

– Виртуальные адреса транслируются в адреса в основной памяти

– Основная память может быть расширена на вторичную

– Размер памяти ограничен схемой адресации, но не фактическим количеством ячеек

Организация виртуальной памяти это совокупность аппаратных и программных средств

**Схемы реализации**

— совокупность страничной и сегментной адресации

– Логические адреса динамически транслируются в физические

 – Сегменты и страницы не должны располагаться последовательно в основной памяти

– Любая часть процесса в разные моменты выполнения может находится во вторичной памяти и менять физические адреса в основной

**Resident set (резидентная часть)** — часть процесса, находящаяся в основной памяти **Real memory** — часть памяти, где процесс непосредственно выполняется

**Virtual memory** — часть памяти, которую процесс может занять

Следовательно, ОС может одновременно с бОльшим количеством процессов

**虚拟地址空间 - 具有虚拟地址的单个进程的区域**

**虚拟内存是内存中进程的布局，其中：**

**- 辅助存储器的寻址方式与主存储器相同**

**– 虚拟地址被翻译成主存中的地址**

**– 主存储器可以扩展到辅助存储器**

**– 内存大小受寻址方案的限制，但不受实际单元数的限制**

**虚拟内存的组织是硬件和软件的结合**

**实施方案**

**- 页和段寻址的组合**

**– 逻辑地址被动态转换为物理地址**

**– 段和页在主存中不能是连续的**

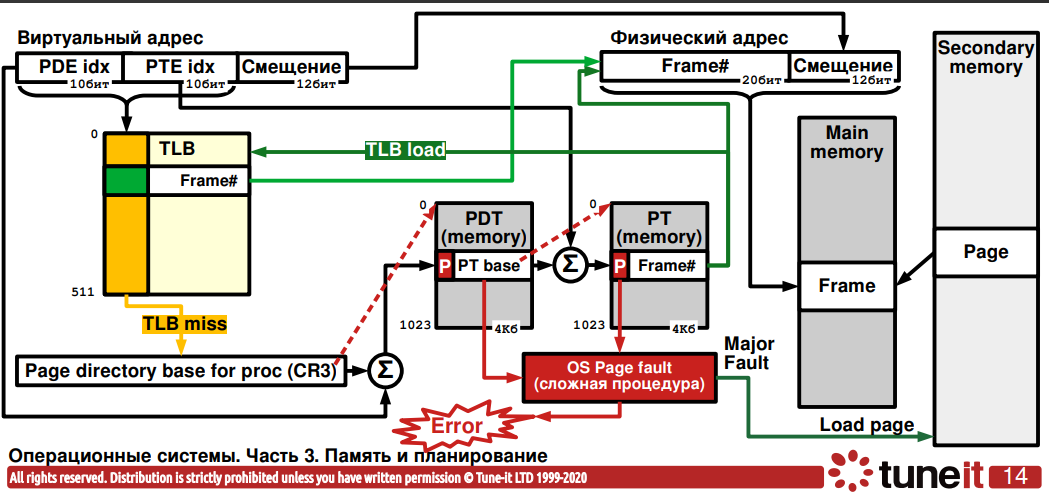
**– 进程在不同执行时刻的任何部分都可以在辅助内存中，并在主内存中更改物理地址**

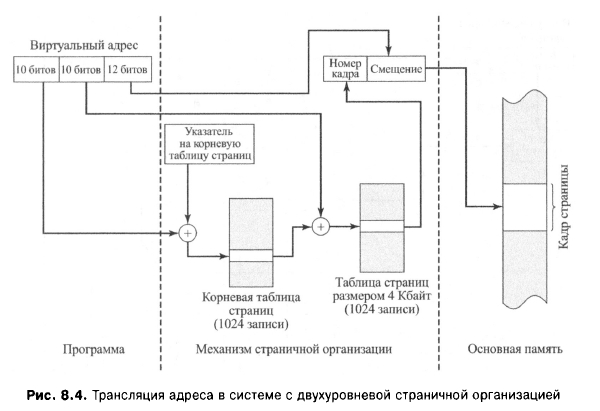
**驻留集（resident part）——位于主内存中的部分进程 Real memory——直接执行进程的内存部分**

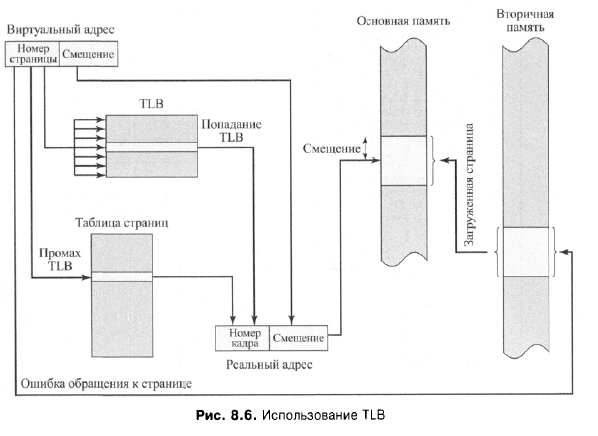
**虚拟内存 - 进程可以占用的内存的一部分**

**因此，操作系统可以同时处理更多的进程**

44. Виртуальный страничный обмен. Двухуровневая организация MMU и TLB 80386. (для КОТ и ГТ — общие принципы)        虚拟寻呼。 MMU 和 TLB 80386 的两级组织。（对于 KOT 和 GT - 一般原则）







TLB的两级虚拟内存.

虚拟内存386处理器: 虚拟地址被分成三部分,由于表是两级,所以有两个索引.

 PDE索引(页目录项)表示在PDT(页目录表--主进程放置表中的偏移量,也就是其他映射的目录).

PTE(页表条目)索引从PT(页表)中获取数据,从中获取主存储器感兴趣帧的编号.

虚拟地址中还有一个12位的偏移量,这决定了每个页面和内存帧都会占用4kb. 将偏移量复制到物理地址.

该寄存器对每个进程都有唯一含义,指向PDT开头.对于每个进程,该寄存器存储在进程的上下文中.因此页表上下文切换,页表以相同方式改变. 一个进程使用一组表,但是另一个进程使用一组不同的表.

TLB是唯一一个关联内存基础上的设备.

取两个指标PDE和PTE,然后,一次在所有单元格中执行搜索. 这种情况下的PDE和PTE索引集是一个标签. TLB的输出包含根据虚拟地址进行换转的帧号.因此,由寄存器定义一组虚拟地址始终对应唯一的帧号,随后将向其添加偏移量. 之后出现物理地址, 如果程序访问内存地址,并且地址已经在TLB中,则帧的编号带入物理地址.

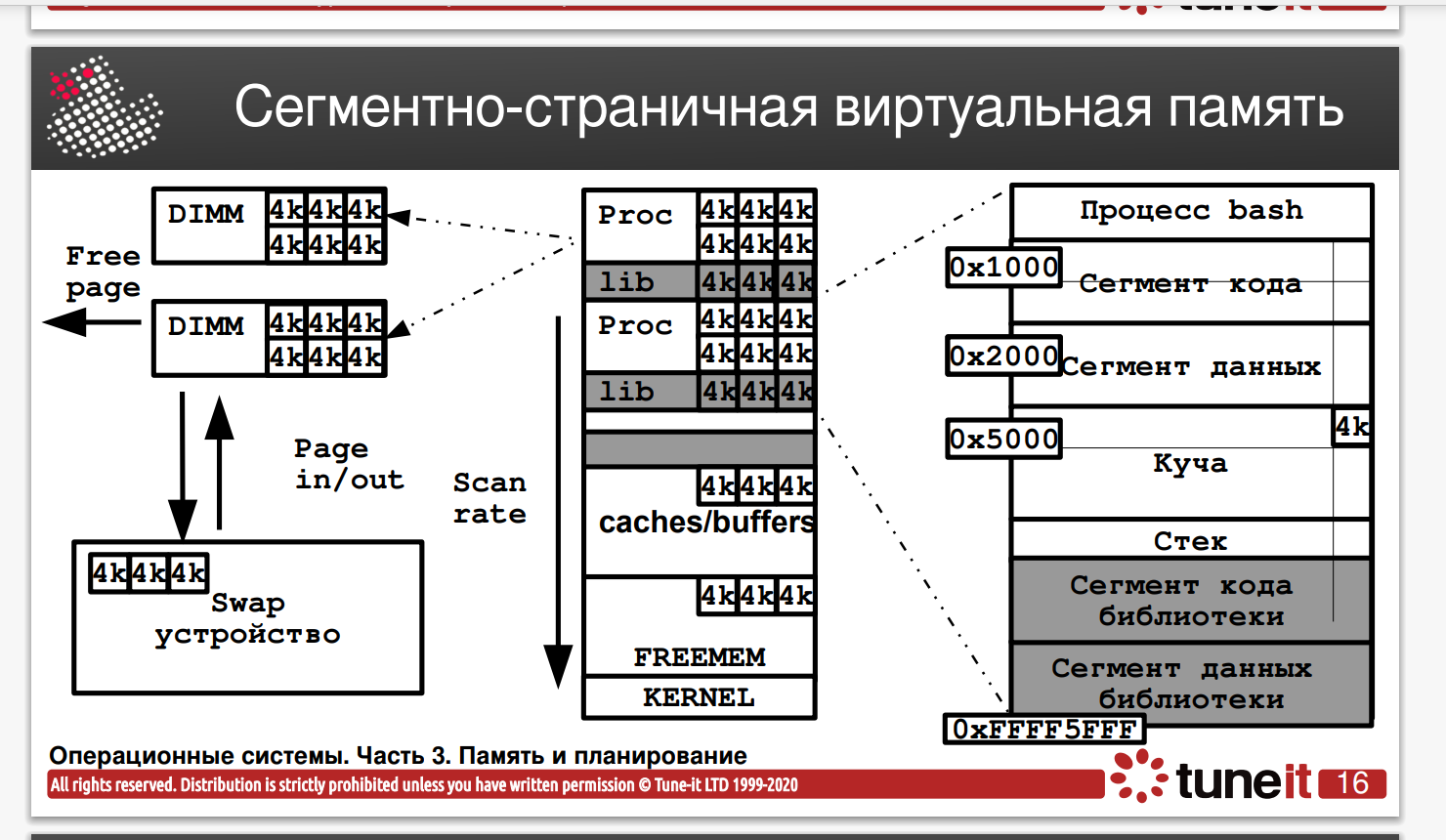
内存是关联的,它实际在一个时钟周期内搜索所有这些单元格.  TLB始终包含与所需单元格对应的条目.  实现TLB的设备在处理器中, PT和PDT在内存中. TLB是在硬件中实现的,位于处理器中作为MMU(内存管理单元)的一部分.

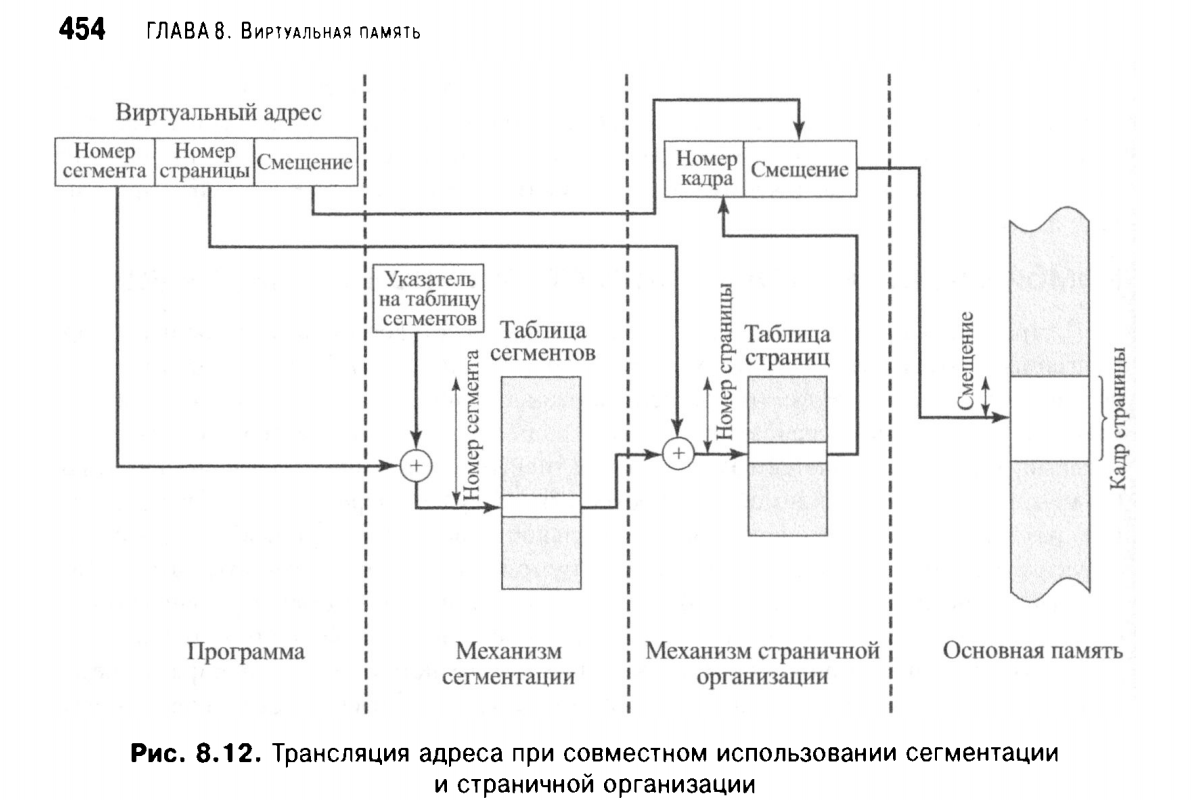
PDT和PT表可能会出现表不存在内存中,也就是被卸载且内存不足.

缺少记录可以调用一个进程,在内核中成为页面错误-访问页面失败.

46. Сегментно-страничная виртуальная память.

segment-page虚拟内存。





虚拟内存是操作系统里的概念，对操作系统来说，虚拟内存就是一张张的对照表，进程获取内存的数据需要去对应的物理内存的地址里面找,通过虚拟内存机制，每个进程都以为自己占用了全部内存，进程访问内存时，操作系统都会把进程提供的虚拟内存地址转换为物理地址，再去对应的物理地址上获取数据。CPU 中有一种硬件，内存管理单元 MMU专门用来将翻译虚拟内存地址。

虚存寻址 linux分为三级页表结构 由 页目录, 页中间目录,页表组成.

操作系统虚拟内存到物理内存的映射表，就被称为页表。操作系统将内存以4K为单位分为各个页,那么虚拟内存页对应物理内存页的映射表就大大减小了.