

Lecture 6: 知识 - 内存管理基础

考试大纲

(一)内存管理基础

- 内存管理的基本概念：
 - 程序装入与链接；
 - 逻辑地址与物理地址空间；
 - 内存保护；
 - 内存分配与回收
- 连续分配管理方式
- 分页管理方式
- 分段管理方式
- 段页式管理方式

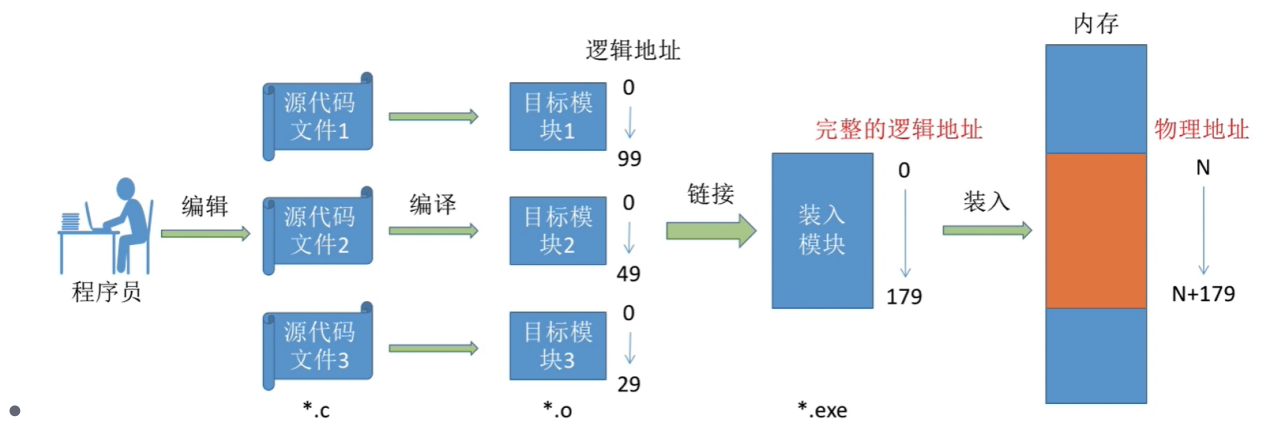
内存管理的基本概念

概念：基本概念

- 概念：内存
 - 是用于存放数据的硬件，程序执行前需要**先放到内存**中才能被 CPU 处理；
 - 为了区分不同的程序、数据，需要为存储单元编其地址；
- 概念：存储单元
 - 如果计算机按字节编址 则每个存储单元大小为 1 字节，即 $1B = 8b$ ；
 - 如果字长为 16 位的计算机按字编址，则每个存储单元大小为 1 个字；
- 概念：MMU
 - 真题：2024年
 - 内存管理单元 MMU (Memory Management Unit) 的主要功能是虚实地址转换、内存保护和 Cache 等，是一种负责处理中央处理器的内存访问请求的计算机硬件；

概念：两种地址

- 相对地址
 - 又称逻辑地址
- 绝对地址
 - 又称物理地址
- 图示：



程序的装入与链接

编译

概念：编译 (Compilation)

- 开始时，程序员编写的高级语言源代码（如 C 或 C++）需要被编译器转换成机器可以理解的代码。
- 编译器首先进行预处理，处理源代码中的宏和条件编译指令。
- 然后，编译器进行词法分析和语法分析，将源代码转换成抽象语法树（AST）。
- 接下来，编译器进行语义分析和中间代码生成，将AST转换成特定于编译器的中间表示形式。
- 最后，编译器优化代码并将其转换成目标代码，通常是汇编语言。

汇编

概念：汇编

- 汇编器将编译器生成的汇编代码转换成机器代码。
- 这个过程称为汇编，它将汇编指令映射到具体的机器指令上。

链接

概念：链接

- 链接器将汇编生成的目标文件（object files）与其他库文件或目标文件合并，生成一个单一的可执行文件；
- 链接器解决程序中的符号引用，将它们转换为逻辑地址；

概念：链接的 3 种方式

- 1. 静态链接：
 - 在程序运行之前，先将各目标模块及它们所需的库函数连接成一个完整的可执行文件（装入模块），之后不再拆开；
- 2. 装入时动态链接：
 - 将各目标模块装入内存时，边装入边链接的链接方式；
- 3. 运行时动态链接：
 - 在程序执行中需要该目标模块时，才对它进行链接；

- 优点：便于修改和更新，便于实现对目标模块的共享；

装入

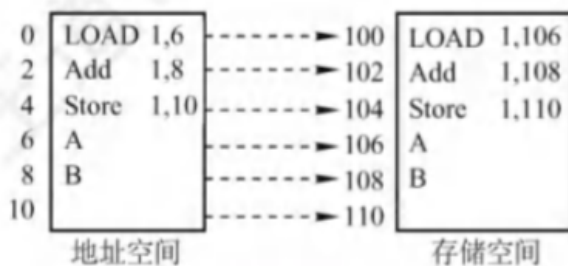
绝对装入

- 概念：
 - 在编译时，如果知道程序将放到内存中的哪个位置，编译程序将产生**绝对地址的目标代码**；
 - 即：程序中的指令具有绝对内存地址。装入时不需要修改程序中的地址信息；
 - 编译、链接后得到的装入模块的指令直接就使用了绝对地址
- 特点
 - 只适用于**单道程序环境**；
 - 缺乏平台独立性。依赖于其在具体平台的地址位置；
- 应用：
 - 简单的嵌入式系统可能会使用绝对装入，因为它们在编译时就确定了程序将运行的内存地址；

静态重定位

- 概念：
 - 又称可重装定位装入；
 - 程序或模块在**编译时**不分配具体的内存地址，而是在**装入时**根据当前内存情况分配地址，在**编译后**再将当前程序的地址填上合适的位置；
 - 编译、链接后的装入模块的地址是从 0 开始的，指令中使用的地址、数据存放的地址都是**相对于起始地址而言的逻辑地址**；
 - 可根据内存的当前情况，将装入模块装入到内存的适当位置；
 - **装入时对地址进行重定位**，将逻辑地址变换为物理地址（地址变换是在装入时一次完成的）
- 特点
 - 1. 在一个作业装入内存时，必须分配其要求的**全部内存空间**；
 - 2. 如果没有足够的内存，就不能装入该作业；
 - 3. 作业一旦进入内存后，在运行期间就**不能再移动**，也**不能再申请**内存空间；

• 图示：

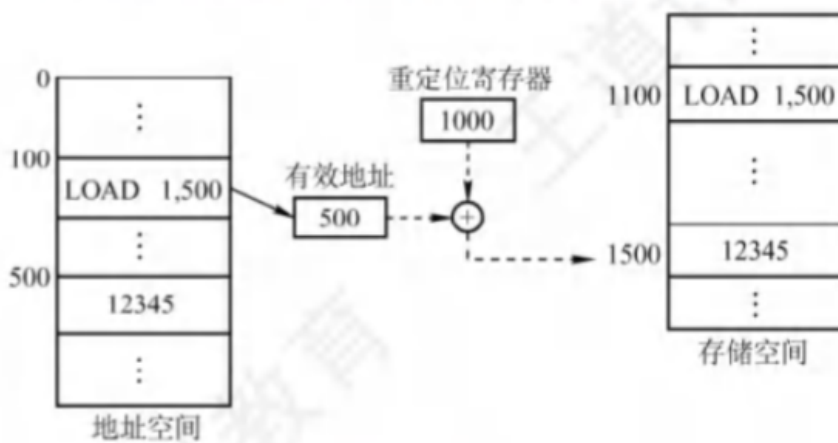


(a) 静态重定位

动态重定位

- 概念：

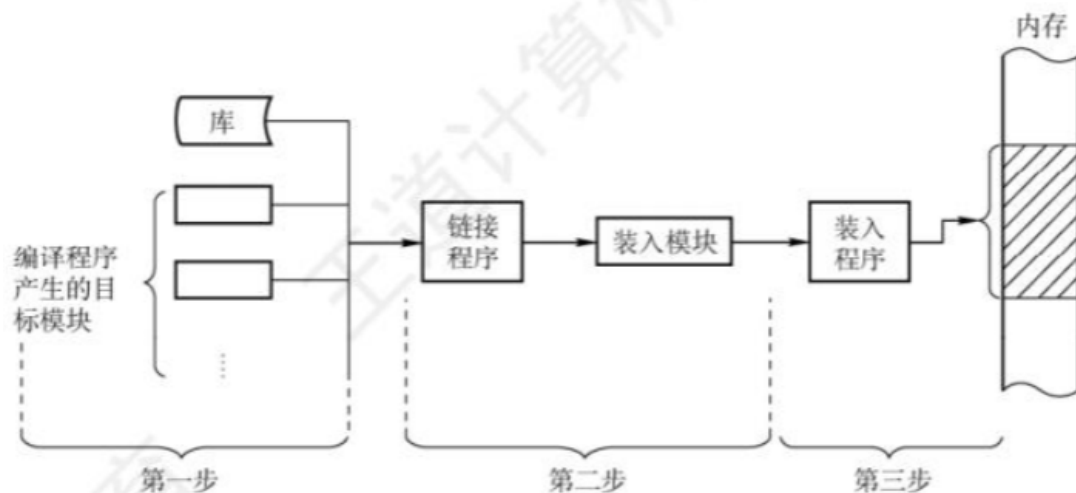
- 又称动态运行时装入；
- 编译、链接后的装入模块地址都是从 0 开始的；
- 装入程序把装入模块装入内存后，并不会立即把逻辑地址转换为物理地址，而是把地址转换推迟到**程序真正要执行时才进行**；
- 程序在执行前不完全加载到内存中，而是在需要时动态加载部分代码或数据。
- 因此装入内存后所有的地址依然是逻辑地址，这种方式需要一个 **重定位寄存器** 的支持；
- 特点
 - 1. 允许程序在内存中发生移动；
 - 2. 可以把程序 **分配到不连续的存储区**，可以将程序分配到不连续的存储区，在程序运行前只需装入它的部分；
 - 3. 运行时可以动态的申请分配内存；
 - 4. 可以为用户提供一个比存储空间大的空间的地址空间；
- 应用：
 - 某些现代操作系统和服务，如 **Java** 的 **JVM**（Java 虚拟机），使用类加载器在运行时动态加载 **Java** 类；
- 图示：



(b) 动态重定位

程序装入与链接分析

概念：装入与链接的过程



概念：装入与链接的区别

装入（Loading）和链接（Linking）是两个不同的概念，它们发生在程序的生命周期中的不同阶段：

1. 装入（Loading）：

- 装入是操作系统将程序从磁盘或其他存储介质加载到内存中的过程。
- 当程序被执行时，操作系统会找到程序的可执行文件，将其复制到内存中的一个适当位置。
- 装入可能包括代码段、数据段和堆栈的加载。
- 在现代操作系统中，装入还可能涉及到内存管理技术，如分页或分段。

2. 链接（Linking）：

- + 链接是将程序的各个部分（通常是编译后生成的目标文件）组合成一个单一的可执行文件的过程。
- + 链接发生在程序编译阶段之后，可能在程序执行之前或作为执行过程的一部分。
- + 链接分为两种类型：
 - + **静态链接**：在程序执行前，静态链接器将程序的所有模块和它们所需的库函数链接在一起，形成一个单一的可执行文件。
 - + **动态链接**：程序在执行时，动态链接器（或运行时链接器）将程序的模块和它们所需的库函数链接在一起。这允许共享内存中的库代码，并在不同程序之间重用。

区别：

- **时间**：装入发生在程序执行时，而链接可以发生在程序执行前（静态链接）或执行时（动态链接）
- **目的**：装入的目的是将程序加载到内存中以便执行；链接的目的是将程序的不同部分或库函数组合成一个完整的程序

概念：地址转换发生阶段

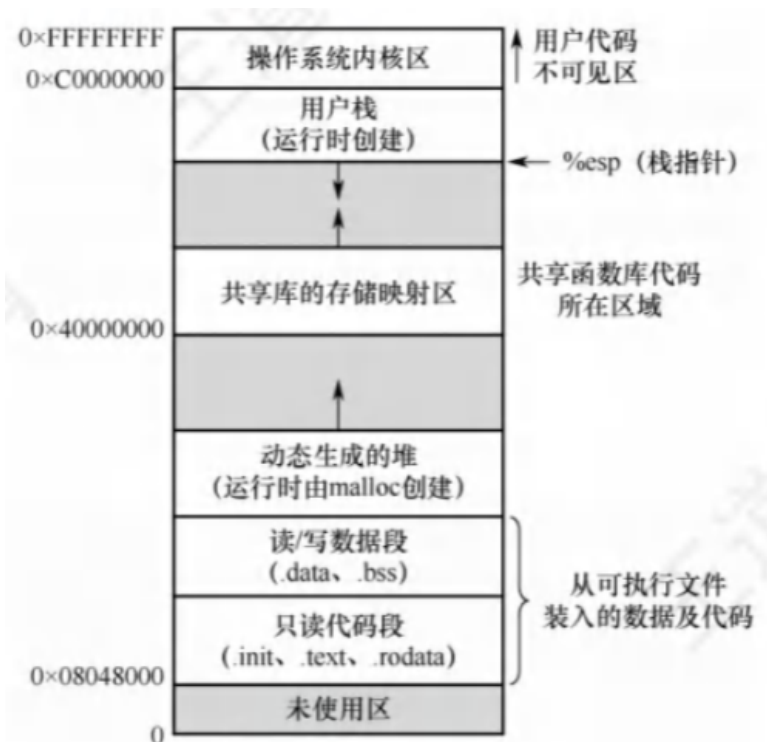
- **真题：**2011年
- **编译时**：编译器生成的代码可能包含地址的**占位符**，这些地址在编译时是未知的；
- **链接时**：链接器为代码和数据分配**逻辑地址**，但这些地址还不是最终的物理地址；
- **装入时**：装入程序根据操作系统的内存管理策略，将逻辑地址映射到**物理地址**；

逻辑地址与物理地址空间

概念：进程的内存映像

- 概念：
 - 代码段：
 - 即程序的二进制代码，代码段是只读的，可以被多个进程共享；
 - 数据段：
 - 即程序运行时加工处理的对象，包括全局变量和静态变量；
 - 进程控制块（PCB）：
 - 存放在系统区；
 - 操作系统通过 PCB 来控制和管理进程；
 - 堆：

- 用来存放动态分配的变量；
- 通过调用 `malloc` 函数动态地向高地址分配空间
- 栈：
 - 用来实现函数调用；
 - 从用户空间的**最大地址往低地址**方向增长；
- 解释：
 - 代码段与数据段：
 - 代码段和数据段在程序调入内存时就指定了大小；
 - 堆和栈：
 - 堆：
 - 当调用像 `malloc` 和 `free` 这样的 C 标准库函数时，堆可以在运行时动态地扩展和收缩；
 - 用户栈：
 - 在程序运行期间也可以动态地扩展和收缩，每次调用一个函数，栈就会增长。从一个函数返回时，就会收缩；
 - 共享库：
 - 用来存放进程用到的共享函数库代码，如 `print` 函数等；
 - 在只读代码段中：
 - `.init` 是程序初始化时调用的 `init` 函数；
 - `.text` 是用户程序的机器代码；
 - `.rodata` 是只读数据；
 - 在读/写数据段中：
 - `.data` 是已初始化的 全局变量 和 静态变量；
 - `.bss` 是未初始化及所有初始化为 0 的全局变量和静态变量。
 - 图示：
 - 32 位下的 OS，虚拟地址空间从 $0 \sim 2^{32}$ 位大小，但其中只有一小部分是数据的；
 - PC 会指向下一条指令的虚拟地址；



分析：发生虚拟地址 $\rightarrow$ 物理地址转换的两个时机

- 取指令时：
 - 在取指令时，需要从 **PC** 当中指令所在的虚拟地址（**PC** 指向的是要执行的下一条指令的虚拟地址）；
 - 取出此虚拟地址后，需要进行查页表、获得其物理地址，进而需要到主存中进行访问，取出指令，并计算下一条指令的虚拟地址；
- 执行指令时：
 - 指令当中可能包含有数据，有些是间接寻址、有些是直接寻址、有些是立即数寻址，但无论是哪种指令，其存储的如果是地址时、一定是虚拟地址；因此执行指令可能会涉及访问主存的操作；
 - 执行的指令会放在 **IR** 寄存器当中，指令当中会有操作码以及地址码字段，此时可以根据操作码字段中的内容，知道当前地址码字段的值所表示的含义是间接寻址、直接寻址等方式；

内存保护

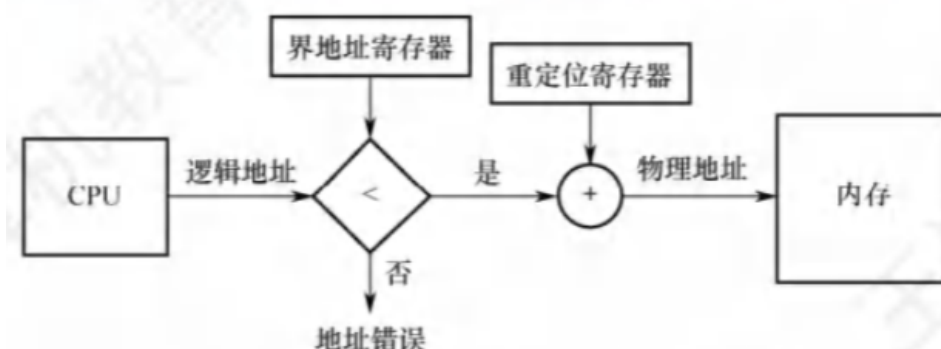
概念：什么是内存保护

- 目的：防止 **CPU** 访问被禁止访问的区域；

概念：内存保护 2 种方法

- 方法一：界地址保护
 - 概念：
 - 在 **CPU** 中设置一对**上、下限寄存器**，存放进程的上，下限地址。
 - 进程的指令要访问某个地址时，**CPU** 检查是否越界；
 - 真题：** 2009年
 - 分区分内存管理方式的主要保护手段就是界地址保护；

- 每个进程都拥有自己独立的进程空间，若一个进程在运行时所产生的地址在其地址空间之外，则发生地址越界，因此需要进行地址保护；
- 即当程序要访问某个内存单元时，由硬件检查是否允许，若允许则执行，否则产生地址越界中断；
- 方法二：越界检查
 - 概念：
 - 采用重定位寄存器(又称为**基址寄存器**)和界地址寄存器(又称限长寄存器)进行越界检查；
 - 重定位寄存器中存放的是**进程的起始物理地址**；
 - 界地址寄存器中存放的是**进程的最大逻辑地址**；
 - 实现内存保护需要重定位寄存器和界地址寄存器，因此要注意两者的区别。重定位寄存器是用来“加”的，逻辑地址加上重定位寄存器中的值就能得到物理地址；界地址寄存器是用来“比”的，通过比较界地址寄存器中的值与逻辑地址的值来判断是否越界；
 - 加载重定位寄存器和界地址寄存器时必须使用**特权指令**，只有操作系统内核才可以加载这两个存储器。这种方案允许操作系统内核修改这两个寄存器的值，而不允许用户程序修改；
 - 分析：
 - 内存管理部件将逻辑地址与界地址寄存器进行比较，若未发生地址越界，则加上重定位寄存器的值后映射成物理地址，再送交内存单元；



概念：内存共享

- 并不是所有的进程内存空间都适合共享，只有那些只读的区域才可以共享；
- 可重入代码也称纯代码，是一种允许**多个进程同时访问但不允许被任何进程修改**的代码。但在实际执行时，也可以为每个进程配以局部数据区，将在执行中可能改变的部分复制到该数据区，这样，程序在执行时只需对该私有数据区中的内存进行修改，并不去改变共享的代码；
- 下面通过一个例子来说明内存共享的实现方式：
- 考虑一个可以同时容纳 40 个用户的多用户系统，他们同时执行一个文本编辑程序，若该程序有 160 KB 代码区和 40 KB 数据区，则共需 8000 KB 的内存空间来支持 40 个用户。若 160 KB 代码是可分享的纯代码，则不论是在分页系统中还是在分段系统中，整个系统只需保留一份副本即可，此时所需的内存空间仅为 $40 \text{ KB} \times 40 + 160 \text{ KB} = 1760 \text{ KB}$ ；
- 对于分页系统，假设页面大小为 4 KB，则代码区占用 40 个页面、数据区占用 10 个页面，为实现代码共享，应在每个进程的页表中都建立 40 个页表项，它们都指向共享代码区的物理页号。此外，每个进程还要为自己的数据区建立 10 个页表项，指向私有数据区的物理页号。对于分段系统，由于是以段为分配单位的，不管该段有多大，都只需为该段设置一个段表项（指

向共享代码段始址，以及段长 160 KB)。由此可见，段的共享非常简单易行。此外，在第 2 章中我们介绍过基于共享内存的进程通信，由操作系统提供同步互斥工具。

内存的分配与回收

概念：内存空间的分配与回收

- 1. 连续分配管理方式
 - (1) 单一连续分配
 - (2) 固定分区分配
 - (3) 动态分区分配
 - 顺序搜索的分配算法
 - 索引搜索的分配算法
- 2. 非连续分配管理方式
 - (1) 基本分页存储管理
 - (2) 基本分段存储管理
 - (3) 段页式存储管理

连续分配管理方式

两种分配方式

- 连续分配：指为用户进程分配的必须是一个连续的内存空间；
- 非连续分配：为用户进程分配的不需要是一个连续的内存空间，可以是离散的；

内部碎片与外部碎片

- 内部碎片：分配给某进程的内存区域中，如果有些部分没有用上；
 - 比如：采用固定分区分配，每个分区为 10mb 大小。此时要分配一个 8mb 的大小的空间，此时就需要为其分配一个 10mb 大小的分区，因此产生了内部碎片；
- 外部碎片：指内存中的某些空闲分区由于太小而难以利用（紧凑技术解决）；
 - 可以通过紧凑（拼凑，Compaction）技术来解决外部碎片；

单一连续分配

概念：单一连续分配

- 概念：
 - 在单一连续分配方式中，内存被分为系统区和用户区，系统区仅供操作系统使用，通常在低地址部分；
 - 用户区内存中仅有一道用户程序，即用户程序独占整个用户区；
 - 这种方式的优点是简单、无外部碎片：不需要进行内存保护，因为内存中永远只有一道程序。缺点是只能用于单用户、单任务的操作系统中：有内部碎片：存储器的利用率极低；
- 特点
 - 优点：
 - 实现简单；

- **无外部碎片**，可以采用覆盖技术扩充内存
- 不一定需要采取内存保护 (eg: 早期的 PC 操作系统 MS-DOS)
- 缺点:
 - 只能用于**单用户、单任务**的操作系统中
 - **有内部碎片** (分配给某进程的内存区域中, 如果有些部分没有用上, 就是“内部碎片”);
 - 存储器利用率极低
- 补充:

固定分区分配

概念: 固定分区分配

- 含义
 - 为了能在内存中装入多道程序, 且这些程序之间又不会相互干扰, 于是将整个用户空间划分为**若干个固定大小的分区**, 在**每个分区中只装入一道作业**;
- 分类
 - 分区大小相等:
 - 缺乏灵活性, 但是很适合用于一台计算机控制多个相同对象的场合
 - 分区大小不等:
 - 增加了灵活性, 可以满足不同大小的进程需求。
 - 根据常在系统中运行的作业大小情况进行划分;
- 特点
 - 优点:
 - 实现简单, 无外部碎片
 - 缺点:
 - 当用户程序太大时, 可能所有的分区都不能满足需求, 此时不得不采用覆盖技术来解决, 但这又会降低性能;
 - 会**产生内部碎片**, 内存利用率低;

概念: 分区说明表

- 概念:
 - 为了便于分配和回收, 建立一张分区使用表, 通常按分区大小排队, 各表项包括对应分区的始址、大小及状态 (是否已分配);
 - 分配内存时, 便检索该表, 以找到一个能满足要求且尚未分配的分区分配给装入程序, 并将对应表项的状态置为“已分配”;
 - 若找不到这样的分区, 则拒绝分配;
 - 回收内存时, 只需将对应表项的状态置为“未分配”即可;
- 图示:

分区号	大小/KB	起址/KB	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	未分配

动态分区分配

动态分区分配基本概念

概念：动态分区分配（可变分区分配）

- 概念：
 - 动态分区分配也称可变分区分配，是指在进程装入内存时，根据进程的实际需要，动态地为之分配内存，并使分区的大小正好适合进程的需要；
 - 因此，系统中分区的大小和数量是可变的；
 - 如图所示，系统有 64 MB 内存空间，其中低 8 MB 固定分配给操作系统，其余为用户可用内存；
 - 开始时装入前三个进程，它们分别分配到所需的空間后，内存仅剩 4 MB，进程 4 无法装入；
 - 在某个时刻，内存中没有一个就绪进程，CPU 出现空闲，操作系统就换出进程 2，换入进程 4；
 - 由于进程 4 比进程 2 小，这样在主存中就产生了一个 6 MB 的内存块；
 - 之后 CPU 又出现空闲，需要换入进程 2，而主存无法容纳进程 2，操作系统就换出进程 1，换入进程 2；

图示：



- 问题：外部碎片
 - 核心：外部碎片指的是存在于分区的外部的未分配内存，而不是指内存之外的外存空间碎片；
 - 动态分区在开始时是很好的，但是随着时间的推移，内存中会产生越来越多的小内存块，内存的利用率也随之下降，这些小内存块被称为外部碎片，它存在于**所有分区的外部**，与固定分区中的内部碎片正好相对；
 - 外部碎片可通过紧凑技术来克服，即操作系统不时地对进程进行移动和整理。但是，这需要动态重定位寄存器的支持，且相对费时。紧凑过程实际上类似于 Windows 系统中的磁

盘碎片整理程序，只不过后者是对**外存空间**的紧凑；

概念：动态分区分配的内存回收方法 + 空闲分区表与空闲分区链

- 概念：

- 在动态分区分配中，与固定分区分配类似，设置一张空闲分区链(表)，可以按始址排序；
- 分配内存时：
 - 检索空闲分区链，找到所需的分区，若其大小大于请求大小，则从该分区中按请求大小**分割一块空间**分配给装入进程(若剩余部分小到不足以划分，则不需要分割)，余下部分仍然留在空闲分区链中；
- 回收内存时：
 - 系统根据回收分区的始址，从空闲分区链中找到相应的插入点，此时可能出现四种情况：
 1. 回收区与插入点的前一空闲分区相邻，此时将这两个分区合并并修改前一分区表项的大小为两者之和；
 2. 回收区与插入点的后一空闲分区相邻，此时将这两个分区合并，并修改后一分区表项的始址和大小；
 3. 回收区同时与插入点的前、后两个分区相邻，此时将这三个分区合并，修改前一分区表项的大小为三者之和，并取消后一分区表项；
 4. 回收区没有相邻的空闲分区，此时应该为回收区新建一个表项，填写始址和大小，并插入空闲分区链。
 - 以上内存分区管理方法有一个共同特点，即用户程序在主存中都是**连续存放**的；

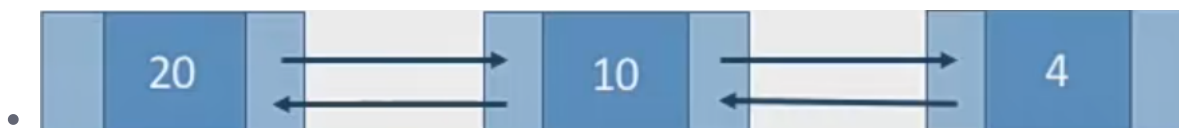
- 空闲分区表：

- 概念：
 - 每个空闲分区对应一个表项。表项中包含分区号、分区大小、分分区起始地址等信息；
- 注意：
 - 各表项的顺序不一定按照地址递增顺序排列，具体的排列方式需要依据动态分区分配算法来确定；
- 图示：

分区号	分区大小 (MB)	起始地址 (M)	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

- 空闲分区链：

- 概念：
 - 每个分区的起始部分和末尾部分分别设置前向指针和后向指针。起始部分处还可记录分区大小等信息；
- 图示：

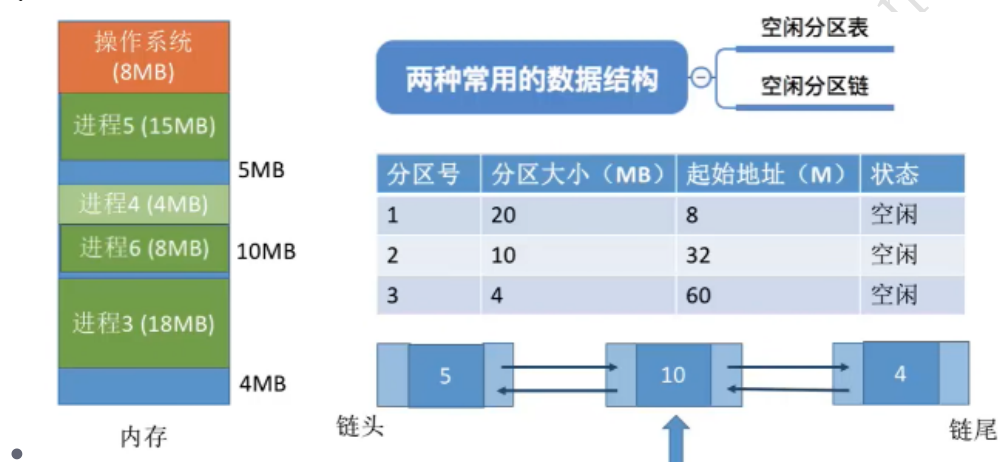


基于顺序搜索的分配算法

首次适合算法

概念：首次适应算法 (First Fit)

- 思想
 - 每次都从**低地址**开始查找，找到第一个能满足大小的空闲分区；
- 实现
 - 空闲分区**以地址递增**的次序排列；
 - 每次分配内存时顺序查找空闲分区链(或空闲分区表)，直到找到大小能满足要求的第一个空闲分区，将其分配给所需进程，然后修改当前表、链的空闲分区的大小数据；
- 图示：



邻近适应算法

概念：邻近适应算法

- 概念：也称循环首次适应算法，由首次适应算法演变而成。不同之处是，分配内存时**从上次查找结束的位置**开始继续查找；
- 邻近适应算法让内存低、高地址部分的空闲分区以同等概率被分配，划分为小分区，导致内存高地址部分没有大空闲分区可用；
- 通常比首次适应算法更差；

最佳适应算法

概念：最佳适应算法 (Best Fit)

- 别名：
 - 最佳适配算法；
- 核心
 - 优先使用**更小的空闲区**，尽可能多地留下大片的空闲区；
 - 空闲分区按容量递增的次序排列；

- 思想
 - 由于动态分区分配是一种连续分配方式，为各进程分配的空闲必须是连续的一整片区域；
 - 因此为了保证当**大进程**到来时能有连续的大片空间，可以尽可能多地留下大片的空闲区，即**优先使用更小的空闲区**；
- 实现
 - 空闲分区按**容量递增次序**链接；
 - 每次分配内存时顺序查找空闲分区链(或空闲分区表)，找到大小能满足要求的**第一个**空闲分区；
- 特点：
 - 最佳适应算法虽然称为最佳，能更多地留下大空闲分区，但性能通常很差，因为每次分配会留下越来越多很小的难以利用的内存块，进而产生最多的外部碎片；
- 真题：2019年
 - 最佳适应算法是指将空闲分区按大小**升序排列**，依次查找，选择第一个可以容纳程序的空间分配出去；
 - 该方法选取最小的可以容纳程序的空闲分区进行分配，该方式从整体空间的利用效率来看不一定是真正的最佳，因为这样划分容易留下许多小空间分区，其**外部碎片是这四种方法中最多的**；

最坏适应算法

概念：最坏适应算法 (Worst Fit)

- 核心
 - 每次分配时优先使用**最大的连续空闲区**；
- 思想
 - 为了解决最佳适应算法问题：即留下太多难以利用的小碎片
 - 可以在每次分配时**优先使用最大的连续空闲区**，这样分配后剩余的空闲区就不会太小，更方便使用。
- 实现
 - 空闲分区按**容量递减次序**链接；
 - 每次分配内存时顺序查找空闲分区链(或空闲分区表)，找到大小能满足要求的第一个空闲分区；
- 特点：
 - 每次都选最大的分区进行分配，虽然可以让分配后留下的空闲区更大，更可用，但是这种方式会导致**较大的连续空闲区被迅速用完**。
 - 如果之后有**大进程**到达，就没有内存分区可用了。

总结：综合来看，4 种算法中，首次适应算法的效果反而更好；

算法	算法思想	分区排列顺序	优点	缺点
首次适应	从头到尾找适合的分区	空闲分区以地址递增次序排列	综合看性能最好。 算法开销小 ，回收分区后一般不需要对空闲分区队列重新排序	
最佳适应	优先使用更小的分区，以保留更多大分区	空闲分区以容量递增次序排列	会有更多的大分区被保留下来，更能满足大进程需求	会产生很多太小的、难以利用的碎片； 算法开销大 ，回收分区后可能需要对空闲分区队列重新排序
最坏适应	优先使用更大的分区，以防止产生太小的不可用的碎片	空闲分区以容量递减次序排列	可以减少难以利用的小碎片	大分区容易被用完，不利于大进程； 算法开销大 （原因同上）
邻近适应	由首次适应演变而来，每次从上次查找结束位置开始查找	空闲分区以地址递增次序排列（可排列成循环链表）	不用每次都从低地址的小分区开始检索。 算法开销小 （原因同首次适应算法）	会使高地址的大分区也被用完

基于索引搜索的分配算法

概念：使用索引搜索分配算法的原因

- 概念：
 - 当系统很大时，空闲分区链可能很长，此时采用顺序分配算法可能很慢；
 - 因此在大、中型系统中往往采用索引分配算法；
- 思想：
 - 根据其大小对空闲分区分类，对于每类 (大小相同) 空闲分区，单独设立一个空闲分区链，并设置一张索引表来管理这些空闲分区链；
 - 当为进程分配空间时，在索引表中查找所需空间大小对应的表项，并从中得到对应的空闲分区链的头指针，从而获得一个空闲分区；

快速适应算法

概念：快速适应算法

- 概念：
 - 空闲分区的分类根据**进程常用的空间大小**进行划分；
- 过程：
 - 分配过程分为两步：
 - 1. 首先根据进程的长度，在索引表中找到能容纳它的最小空闲分区链表；
 - 2. 从链表中取出第一块进行分配；
- 特点：
 - 优点：
 - 查找效率高；
 - 不产生内部碎片；
 - 缺点：

- 回收分区时，需要有效地合并分区，算法比较复杂，系统开销较大；

伙伴系统

概念：伙伴系统

- 核心：
 - 伙伴系统将内存分为以 2 为的幂的块；
 - 当需要分配内存时，系统会找到最小的 2 的幂次方大小的块分配给进程；
- 概念：
 - 规定所有分区的大小均为 2 的 k 次幂；
 - 分配内存时，假设当前需要分配的大小为 n，设 $2^{i-1} < n < 2^i$ ，即表示 n 所需要的内存大小可以在大小为 2^i 的空闲分区链中寻找；
 - 如果没找到，就到 2^{i+1} 中找空闲的区域；
 - 找到后，将这个区域分成两半，一半分配给当前的 n，另一半作为其伙伴，加入到大小为 2^i 的空闲分区链当中；
 - 若不存在，则继续查找，直至找到为止；
 - 回收时，也可能需要对伙伴分区进行合并；
- 特点：
 - 优点：
 - 分配和回收速度快，减少了外部碎片；
 - 缺点：
 - 可能导致内部碎片，因为分配的内存块可能比需求大；
- 真题：2024年
 - 题目：

下列算法中，每次回收分区时仅合并大小相等的空闲分区的是()。

• A. 伙伴算法 B. 最佳适应算法 C. 最坏适应算法 D. 首次适应算法
 - 答案：
 - A
 - 分析：
 - 伙伴算法通过将空闲内存页面按照 2 的幂次划分，一个大块分裂出来的两个小块，两者互为伙伴；
 - 在伙伴系统中回收空闲块时，只当其**伙伴为空闲块**时才归并成大块。也就是说，若有两个空闲块，即使大小相同且地址相邻，但不是由同一大块分裂出来的，也不归并在一起；

其他索引算法

概念：哈希算法

- 根据空闲分区链表的分布规律，建立哈希函数，构建一张以**空闲分区大小为关键字**的哈希表，每个表项记录一个对应空闲分区链的头指针；
- 分配时，根据所需分区大小，通过哈希函数计算得到哈希表中的位置，从中得到相应的空闲分区链表；

补充：SLABS 分配器 (Simplified Lease Allocator)

- 概念：
 - 和最佳适应算法的思想类似，但是其具体实现是是基于内存池的设计；
 - SLABS 分配器是一种用于操作系统内核的内存分配机制，特别是在 Linux 内核中得到了广泛应用；
 - SLABS 分配器的主要目标是提供一种快速且高效的内核内存分配方法，同时减少内存碎片和提高内存分配的可预测性；
- 对象缓存 (Object Caches)
 - SLABS 分配器使用对象缓存来存储特定类型或大小的对象；
 - 每个缓存可以看作是一个内存池，专门用于分配和释放特定大小的内存块；
- 按需分配
 - 当一个进程请求内存时，SLABS 分配器首先检查相应的对象缓存是否有可用的内存块；
 - 如果有，就直接从缓存中分配；
 - 如果没有，SLABS 分配器会从更大的内存池中获取内存，并将其添加到缓存中；
- 内存池 (Slab Pools)
 - SLABS 分配器将内存分为多个内存池，每个池包含大小相同的内存块 (slabs)；
 - 这些内存池可以针对不同类型的对象进行优化；

非连续分配管理方式

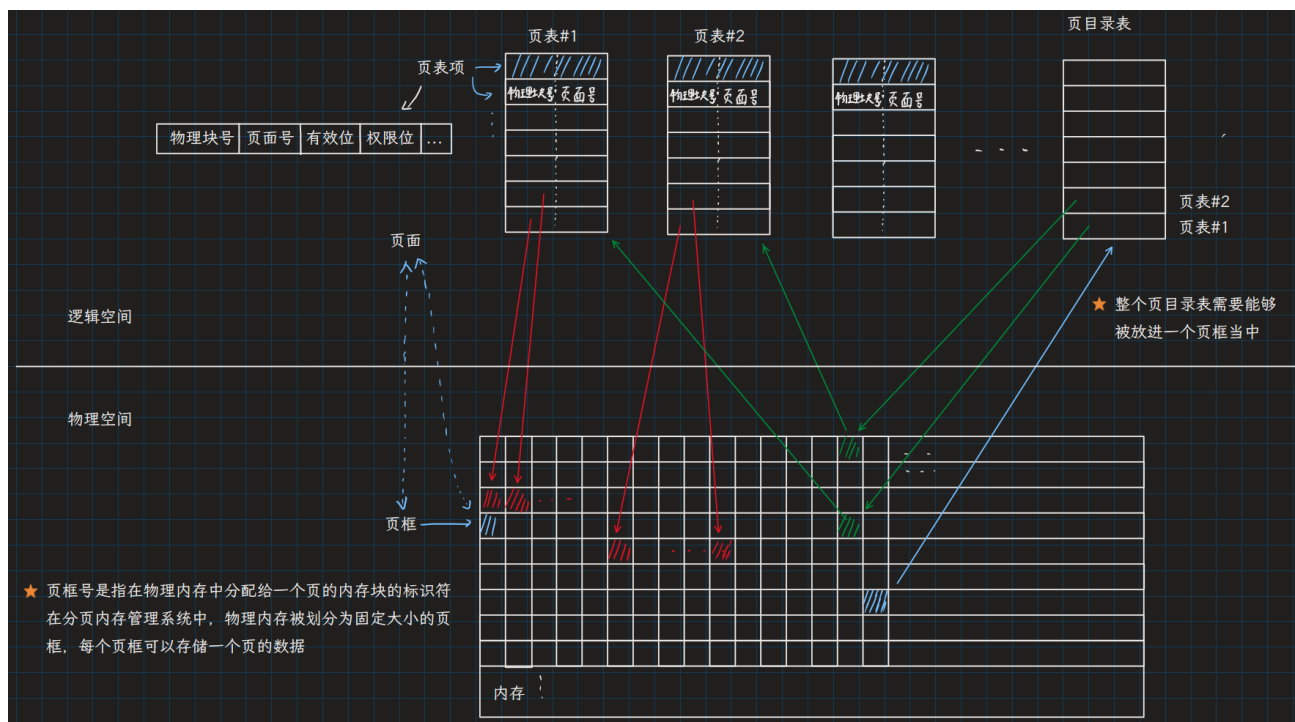
引入：非连续分配方式根据分区的大小是否固定，分为分页存储管理和分段存储管理。在分页存储管理中，又根据运行作业时是否要将作业的所有页面都装入内存才能运行，分为基本分页存储管理和请求分页存储管理；

基本分页存储管理

引入：分页的原因

- 将内存空间分为若干固定大小 (如 4 KB) 的分区，称为页框、页帧或物理块；
- 进程的逻辑地址空间也分为与块大小相等的若干区域，称为页或页面；
- 操作系统以页框为单位为各个进程分配内存空间。从形式上看，分页的方法像是分区相等的固定分区技术，分页管理不产生外部碎片；
- 但它又有本质的不同点：块的大小相对分区要小很多，而且进程也按照块进行划分，进程运行时按块申请主存可用空间并执行；
- 这样，进程只会在为最后一个不完整的块申请一个主存块空间时，才产生主存碎片，所以尽管会产生内部碎片，但这种碎片相对于进程来说也是很小的，每个进程平均只产生半个块大小的内部碎片 (也称页内碎片)；

总结：分页系统中各个概念关系



分页存储基本概念

页框与页面

概念：分页存储

- 概念：
 - 物理空间：内存的页框：
 - 将内存空间分为一个个大小相等的分区，每个分区就是一个“页框”；
 - 每个页框有一个编号，即“页框号”，页框号从 0 开始；
 - 逻辑空间：进程的页面与页号：
 - 页面 $\neq$ 页号；
 - 页号是对于页面的编号；
 - 将进程的逻辑地址空间，也分为与页框大小相等的一个个部分；
 - 每个部分称为一个“页”或“页面”；
 - 每个页面也有一个编号，即“页号”，页号也是从 0 开始；
- 页面与页框关系：
 - 操作系统以页框为单位为各个进程分配内存空间；
 - 进程的每个页面分别放入一个页框中；
 - 进程的页面与内存的页框有一一对应的关系；
- 名词：
 - 页框：
 - 页框 = 页帧 = 内存块 = 物理块 = 物理页面
 - 页框号：
 - 页框号 = 页帧号 = 内存块号 = 物理块号 = 物理页号

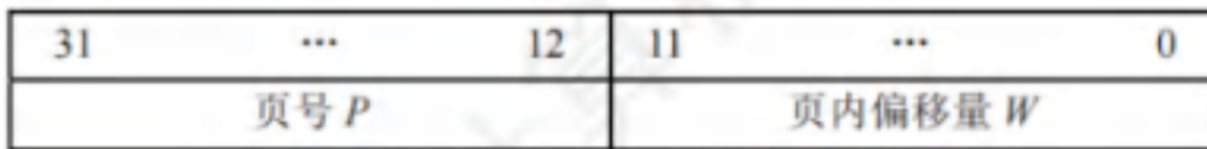
分页存储的逻辑结构

概念：分页存储的逻辑地址结构

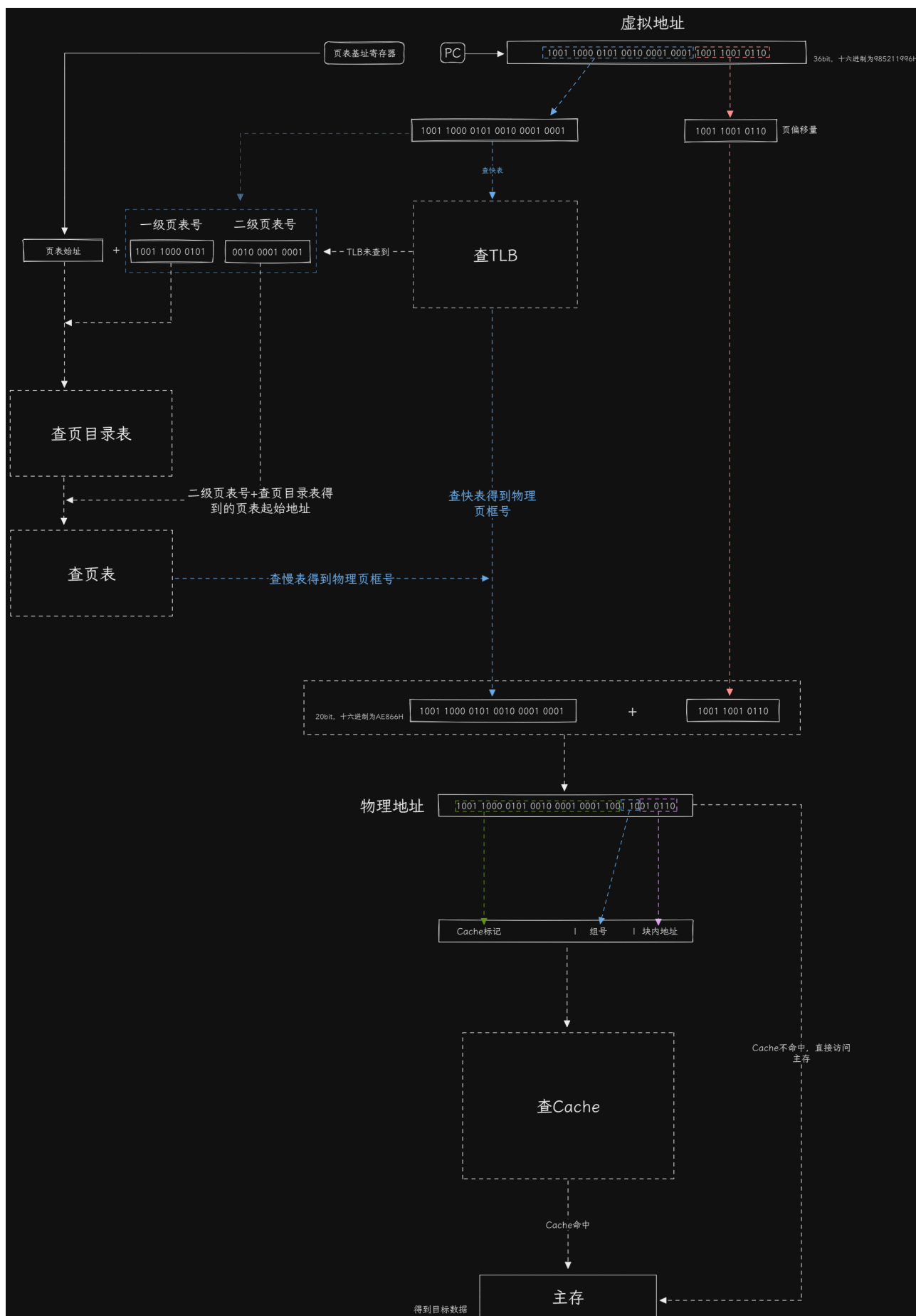
- 概念：
 - 地址结构包含两部分，前一部分为页号 P，后一部分为页内偏移量；
 - 若地址长度为 32 位，其中 0~11 位为页内地址，即每页大小为 $2^{12}B$ ；
 - 12~31 位为页号，即最多允许 2^{20} 页；
- 注意：
 - 地址结构决定了虚拟内存的寻址空间有多大；
 - 在实际问题中，页号、页内偏移、逻辑地址可能是用十进制数给出的，若题目用二进制数给出时，要注意其转换；
- 分析：某两条指令是否属于同一页
 - 题目：
 - 已知当前页面大小为 4KB

int f1 (int n) {			
1	00401000	55	push ebp
	...	...	...
	if (n>1)		
11	00401018	83 7D 08 01	cmp dword ptr [ebp + 8], 1
12	0040101C	7E 17	jle f1+35h (00401035)
	return n*f1(n-1);		
13	0040101E	8B 45 08	mov eax, dword ptr [ebp + 8]
14	00401021	83 E8 01	sub eax, 1
15	同 00401024	50	push eax
16	— 00401025	E8 D6 FF FF FF	call f1 (00401000)
	页 ...	...	...
19	? 00401030	0F AF C1	imul eax, ecx
20	00401033	EB 05	jmp f1+3Ah (0040103a)
	else return 1;		
21	00401035	B8 01 00 00 00	mov eax, 1
	} ...	...	...
26	00401040	3B EC	cmp ebp, esp
	...	...	...
30	0040104A	C3	ret

- 分析：
 - 由于当前页面大小为 4KB，因此页内偏移量为 12 位；
 - 由此可知：低 12 位为页内偏移量，高 20 位为页号，因此 0040104A 和 00401000 的页号都是 00401，属于同一个页面；
- 图示：



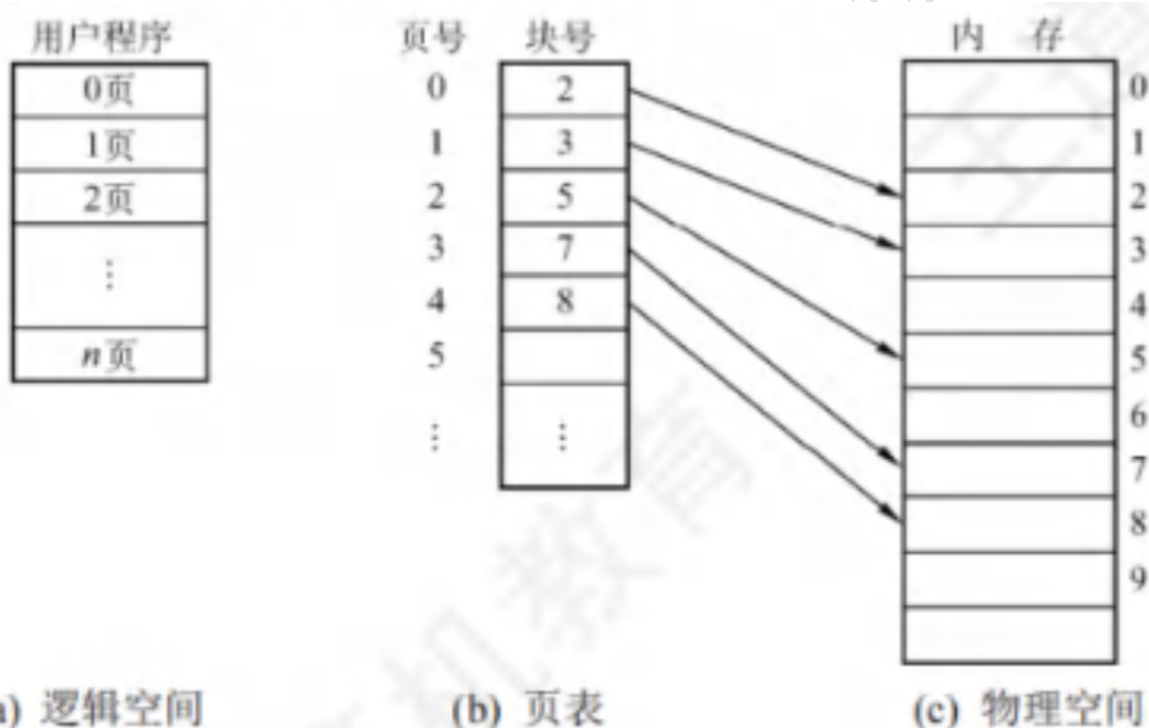
• 分析：转换过程



页表

概念：页表基本概念

- 概念：
 - 为了便于找到进程的每个页面在内存中存放的位置，系统为每个进程建立一张页面映射表，简称页表；
 - 进程的**每个页面对应一个页表项**，每个**页表项由页号和块号**组成，它记录了页面在内存中对应的物理块号；
 - 进程执行时，通过查找页表，即可找到每页在内存中的物理块号。可见，页表的作用是实现从页号到物理块号的地址映射；
- 注意：
 - 页表通常存在 **PCB** (进程控制块)中；
- 图示：



概念：基本地址变换机构与页表寄存器

- 页表寄存器
 - 通常会在系统中设置一个页表寄存器 (**PTR**)，存放页表在内存中的 **起始地址 F** 和 **页表长度 M**；
- 基本地址变换机构
 - 基本地址变换机构可以借助进程的页表，将逻辑地址转换为物理地址；

概念：缺页异常分析

- 真题：2022年
- 概念：
 - 当产生缺页时，缺页异常的处理包括一下步骤：
 1. 将页从外存读入内存中（磁盘 **IO**）；
 2. 建立页号与页框号的对应关系（分配页框+修改页框号）；

- 3. 修改页表中缺页位的**存在位**（修改页表）；
- 注意：
 - 产生缺页时，如果内存中有未使用的页框，即没有存在多余的页时，可以不需要淘汰内存中多余的页；

基本逻辑地址转换

概念：地址转换方式

- 总结：
 - 两次访问内存；
 - 第一次：计算出页号，然后查页表；
 - 第二次：根据页表得到信息，计算出物理地址，访问目标内存单元；
- 计算方法：
 - 假设要访问的逻辑地址 **A**：
 - 1. 分析逻辑地址，确定逻辑地址 **A** 对应的**页号 P** 以及页内偏移量；
 - 公式：根据逻辑地址以及页面大小，计算页号

$$\text{页号} = \frac{\text{逻辑地址}}{\text{页面长度}} (\text{取除法的整数部分})$$

- 公式：根据逻辑地址以及页面大小，计算页内偏移量

$$\text{页内偏移量} = \text{逻辑地址} \% \text{页面长度} (\text{取除法的余数部分})$$

- 即：用逻辑地址除以当前一个页面的大小，整数的部分为页号，余数的部分为页内偏移量；
- 2. 判断页号是否越界，若 **页号 P > 页表长度 M**，则产生越界中断，否则继续执行；
- 3. 在页表中查询页号对应的**页表项**，确定页面存放的物理块号，取出该页表项内容 **b**，即为物理块号；
 - 公式：计算页表项地址

$$\text{页表项地址} = \text{页表始址} F + \text{页号} P * \text{页表项长度}$$

- 4. 计算出物理地址
 - 公式：物理地址计算

$$\text{物理地址} = \text{页面始址} + \text{页内偏移量}$$

- 公式：页面起始地址计算

$$\text{页面起始地址} = \text{内存块号} * \text{块大小} (\text{页面大小})$$

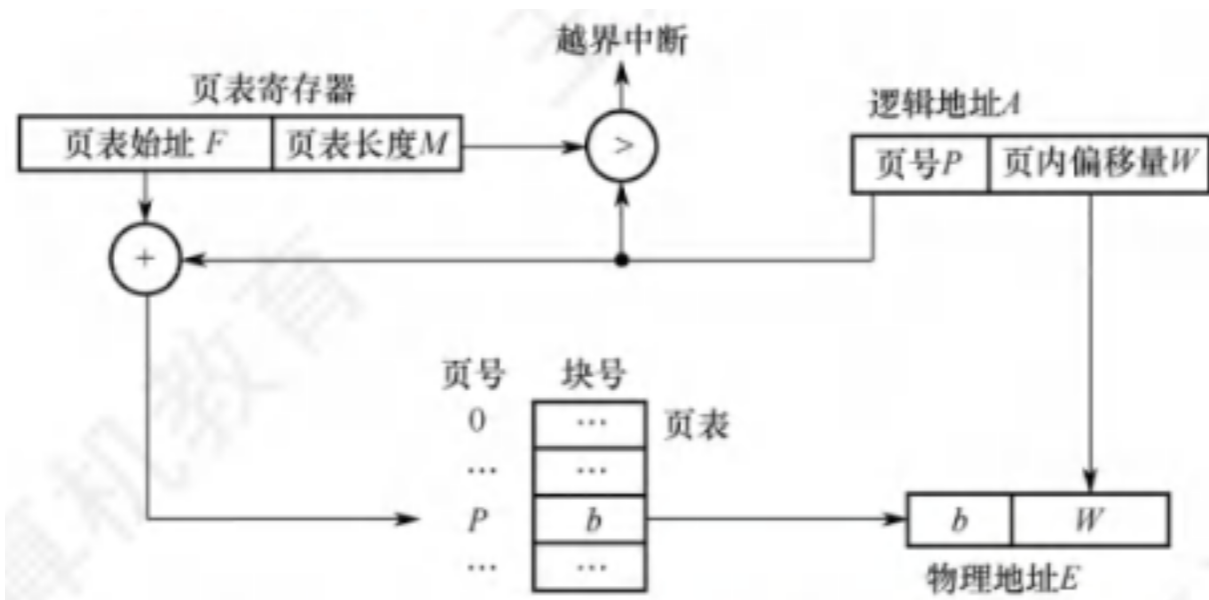
- 举例：
 - 问题：
 - 若页面大小 **L** 为 **1KB**，页号 **2** 对应的物理块为 **b=8**，计算逻辑地址 **A=2500** 的物理地址 **E**；
 - 解答：

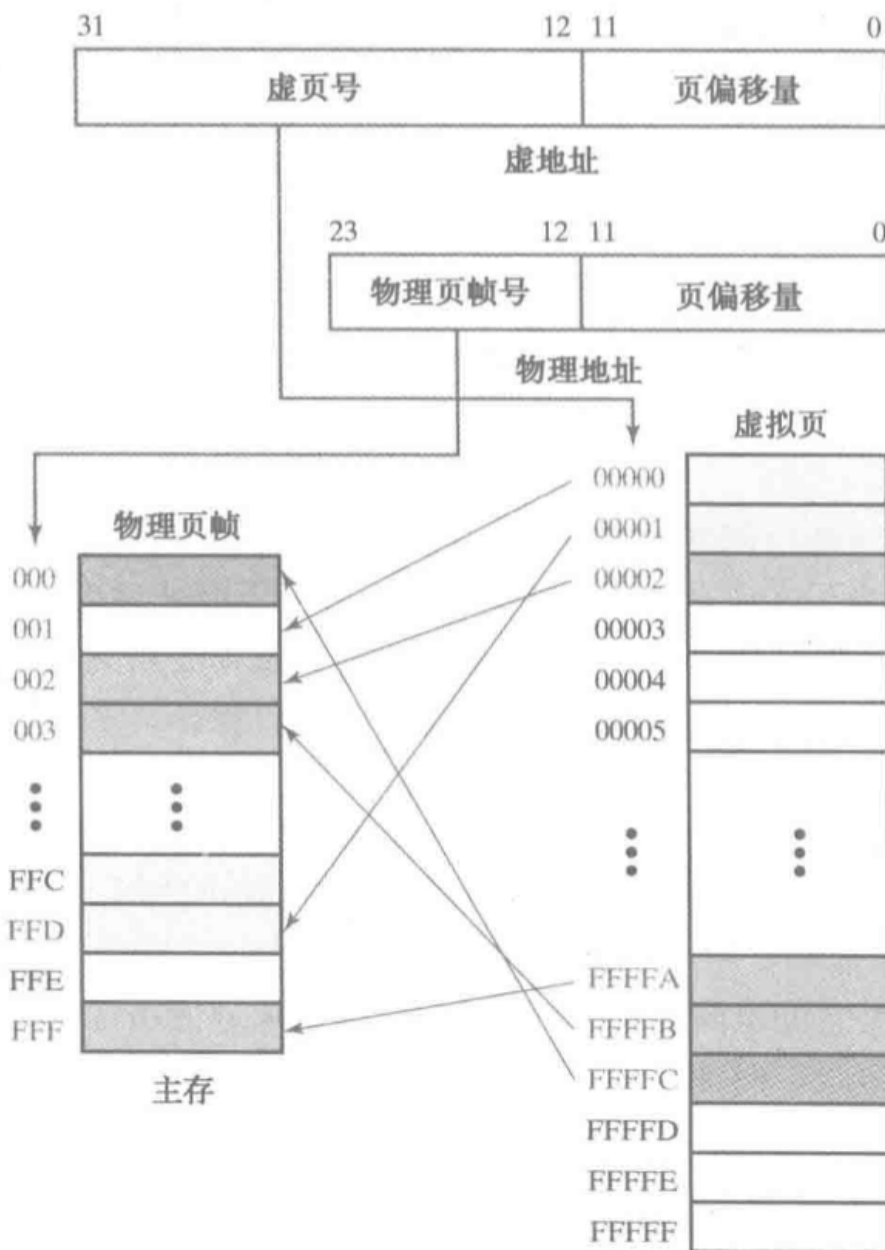
- 由于地址 **A** 的逻辑地址为 **2500**，将其除以页面的大小，得到当前页面的页号：

$$\text{页号} = 2500 / 1024 = 2$$
- 由于知道了其页号为 2，因此可得到其对应的物理块号为 8；
- 知道了物理块号，并且知道页面大小，因此可以计算出其物理地址的起始地址；
- 计算其物理地址：物理地址 = 页面起始地址 + 页内偏移地址 =

$$8 * 1024 + (2500 - 2048) = 8196 + 452 = 8644$$

• 图示：





分析：已知物理内存大小，分析页表项字节数可能大小

• 问题：

- 假设某系统物理内存大小为 **4GB**，页面大小为 **4KB**，则每个页表项至少应该为多少字节？

• 分析：

- 页表项的大小不是随意规定的，而是有所约束的，页表项的作用是找到该页在内存中的位置，因此页表项的大小应该让一个页面能够正好容纳整数个页表项，或便于增加一些其他信息；

• 解答：

- 内存块大小 = 页面大小 = $4K B = 2^{12}B$
- **4 GB** 的内存总共会被分为 $2^{32}/2^{12} = 2^{20}$ 个内存块
- 所以内存块号的范围应该是 $0 \sim 2^{20} - 1$
- 所以内存块号至少要用 **20bit** 来表示
- 即块号至少需要 **3** 个字节 ($3B=24b$)；
- 由于页表式隐含的，因此每个页表项站 **3B**；

• 注意：

- 页表项连续存放，因此**页号可以是隐含的**，不占存储空间（类比数组）；

分析：页表项和页框

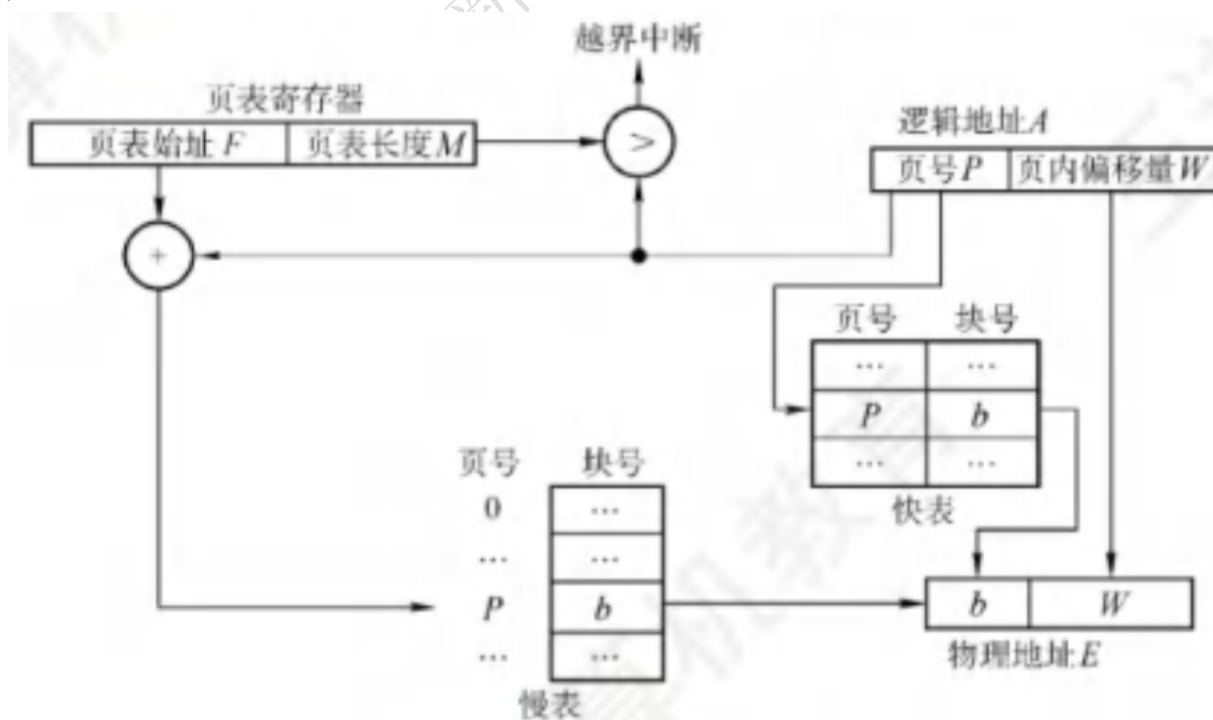
- 概念：
 - 如果每个页表项占 4 字节，则每个页框刚好可存放 1024 个页表项；
 - 1024 号页表项虽然是存放在下一个页框中的，但是它的地址依然可以用 $X + 4 * 1024$ 得出；
- 结论：
 - 理论上，页表项长度为 3B 即可表示内存块号的范围；
 - 但为了方便页表的查询，常常会让一个页表项占更多的字节，使得每个页表项恰好可以装得下整数个页表项；

基于快表的逻辑地址转换

概念：快表

- 概念：
 - 又称联想寄存器（TLB）；
 - 在地址变换机构中增设一个具有并行查找能力的高速缓冲存储器：快表（TLB），用来存放当前访问的若干页表项，以加速地址变换的过程；
 - 与此对应，主存中的页表常称为慢表
 - 即：最近使用过的页表项会放入快表，**访问目标页表项时，先找快表**。如果快表中没有目标页表项，则需要查询内存中的页表；

- 图示：



- 注意：
 1. TLB 不等同于快速缓存。**TLB 中只有页表项的副本，而普通 Cache 中可能会有其他各种数据的副本；**
 2. 不能把整个页表都放入 TLB 中，因为 TLB 虽然快、但是贵、容量小；

概念：快表的地址变换

• 总结：

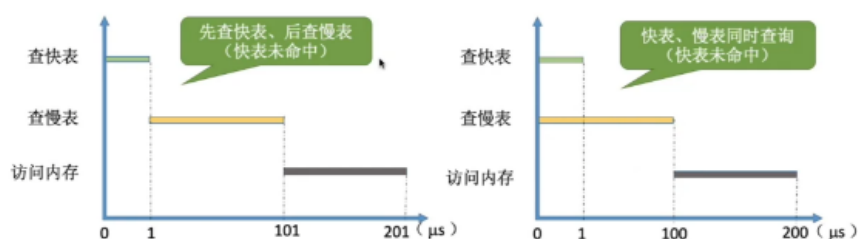
- 在访问目标地址时，先查快表；
- 若在快表中查到，则直接在 TLB 中取出内存块号，然后访问内存 -> 需要访问一次内存；
- 若未在快表中查到，则需要访问内存中的页表，找到页表项后将进行拼接得到物理地址，然后访问内存 -> 需要访问两次内存；

• 概念：

1. CPU 给出逻辑地址后，由某个硬件计算得页号、页内偏移量，将页号与快表中的所有页号进行比较；
2. 如果找到匹配的页号：
 - 要访问的页表项在快表中有副本，则直接从中取出该页对应的内存块号（页框号），再将内存块号（页框号）与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单元。因此若快表命中，则访问某个逻辑地址仅需一次访存即可；
3. 如果在 TLB 中未匹配到页号：
 - 需要访问内存中的页表，找到对应页表项，得到页面存放的内存块号，再将内存块号与页内偏移量拼接形成物理地址；
4. 访问该物理地址对应的内存单元；
 - 若快表未命中，则访问某个逻辑地址需要两次访存
- 注意：
 - 如果未在 TLB 中找到页表项，则需要的主存访问结束后，同时将其页表项存入快表，以便后面可能的再次访问；
 - 但若快表已满，则必须按照一定的算法对旧的页表项进行替换；

• 快慢表同时查询：

• 图示：



分析：TLB 命中平均耗时计算

• 题目：

- 某系统使用基本分页存储管理，并采用了具有快表的地址变换机构。访问一次快表耗时 1us，访问一次内存耗时 100us；
- 若快表的命中率为 90%，那么访问一个逻辑地址的平均耗时是多少？

• 计算：

- 平均耗时： $(1 + 100) * 0.9 + (1 + 100 + 100) * 0.1 = 111us$

总结：基本地址转换与基于快表的地址转换

	地址变换过程	访问一个逻辑地址的访存次数
基本地址变换机构	①算页号、页内偏移量 ②检查页号合法性 ③查页表，找到页面存放的内存块号 ④根据内存块号与页内偏移量得到物理地址 ⑤访问目标内存单元	两次访存
具有快表的地址变换机构	①算页号、页内偏移量 ②检查页号合法性 ③查快表。若命中，即可知道页面存放的内存块号，可直接进行⑤；若未命中则进行④ ④查页表，找到页面存放的内存块号，并且将页表项复制到快表中 ⑤根据内存块号与页内偏移量得到物理地址 ⑥访问目标内存单元	快表命中，只需一次访存 快表未命中，需要两次访存

两级页表

单级页表

分析：单级页表存在的问题

- 问题 1：
 - 所有的页表项都连续存放的基础上才能找到页表项；
 - 因此当页表很大时，需要用很多连续的页框；
- 问题 2：
 - 没有必要让整个页表常驻内存；
 - 因为进程在一段时间内可能只需要访问某个特定的页面；
- 举例：
 - 问题：
 - 若编址单元为 8 位，进程拥有 32 位逻辑地址空间，页面大小 4KB，页表项大小 4B，求其页表项大小；
 - 解答：
 - 因为页面的大小位 4KB，并且以字节编址，因此一个页面内共有 4KB 个存储单元，需要 $\log_2 4K B = 12b$ 来描述页面内的偏移量。因此页内偏移量占 12 位；
 - 因为页内偏移量占 12 位，因此页面号可以占 $32-12=20$ 位；
 - 因为页表项大小为 4B，而页面最多可以有 2^{20} 个，因此可以有 $4B * 2^{20}$ 个页表项；
 - 因为页面大小为 4KB，所以为了存放这些页表项，需要 $4B * 2^{20} / 4K B = 1K$ ，即需要 1K 个页来存储页表项，页表项大小十分巨大。而且因为页表项要求是连续存放于内存中，因此需要一块很大的内存来存放连续的页表项；

解决：把连续存放的页表再分页

- 概念：
 - 可将长长的页表进行分组，使每个内存块刚好可以放入一个分组（比如上个例子中，页面大小 4 KB，每个页表项 4 B，每个页面可存放 1 K 个页表项，因此每 1 K 个连续的页表项为一组，每组刚好占一个内存块，再讲各组离散地放到各个内存块中）

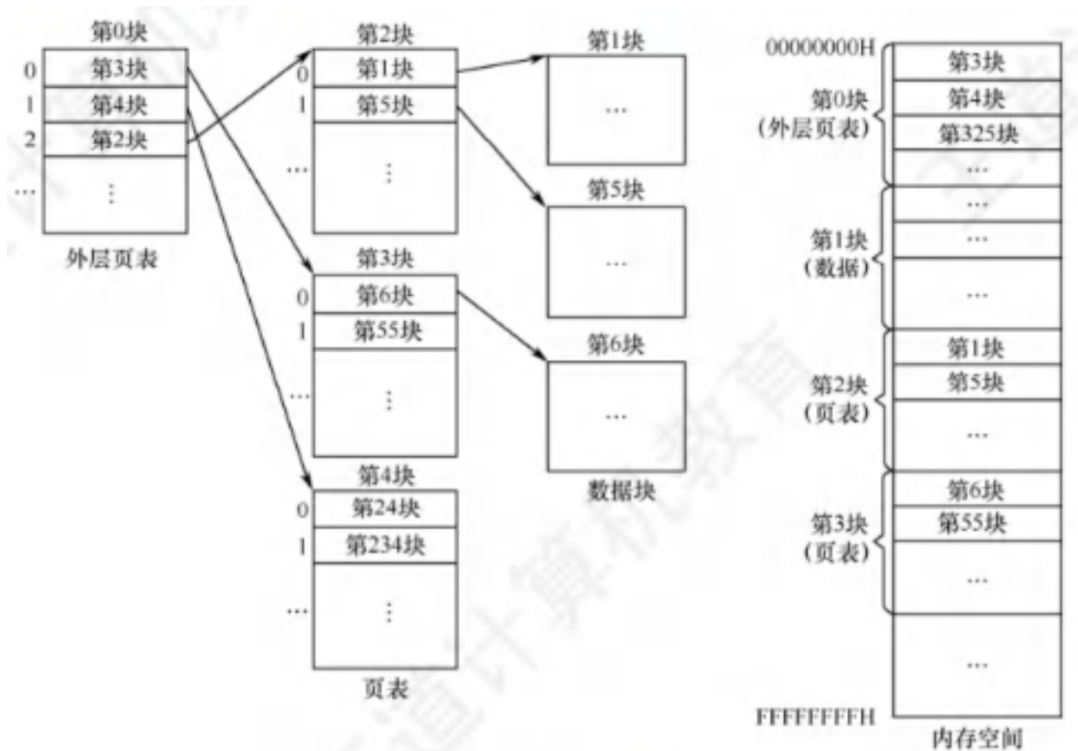
- 另外，要为离散分配的页表再建立一张页表，称为页目录表，或称外层页表，或称顶层页表
- 真题：2014年
 - 多级页表的优点，在于其可以减少页表所占的连续内存空间，但其并不能加快地址变换速度，也不能减少缺页中断的次数；
 - 两级页表需要对各级页表分别进行一次查询，降低了地址变换速度；
 - 页面的缺页与内存容量、内存进程数以及置换算法相关，而多级页表并不能影响页面的缺页，且若多级页表未被装入内存，反而会增加缺页次数；

两级页表

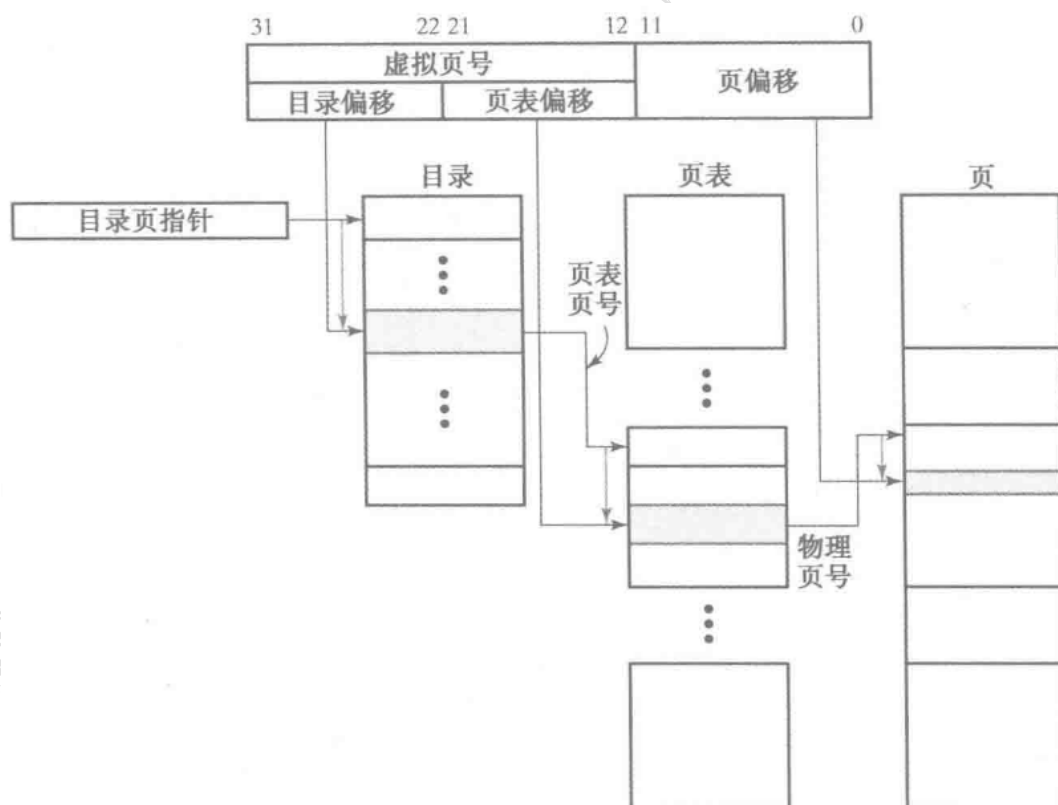
概念：页目录表

- 概念：
 - 1. 对于页表所需的内存空间，采用离散分配方式，用一张索引表来记录各个页表的存放位置，这就解决了页表占用连续内存空间的问题；
 - 2. 只将当前需要的部分页表项调入内存，其余的页表项仍驻留磁盘，需要时再调入（虚拟内存的思想）；
 - 即：为离散分布的页表再建立了一张页表，称之为页目录，或者是外层页表；
- 分析：
 - 在页表的每个表项中，存放的是进程的某页对应的物理块号；
 - 如 0 号页存放在 1 号物理块中，号页存放在 5 号物理块中；
 - 在外层页表的每个表项中，存放的是某个页表分页的始址，如 0 号页表存放在 3 号物理块中。可以利用外层页表和页表来实现进程从逻辑地址到物理地址的变换；
- 注意：
 - 真题：2021年
 - 在系统采用二级页表的情况下，当前页表基址寄存器中的内容为当前进程的一级页表的起始物理地址；
- 图示：
 - 逻辑地址结构：

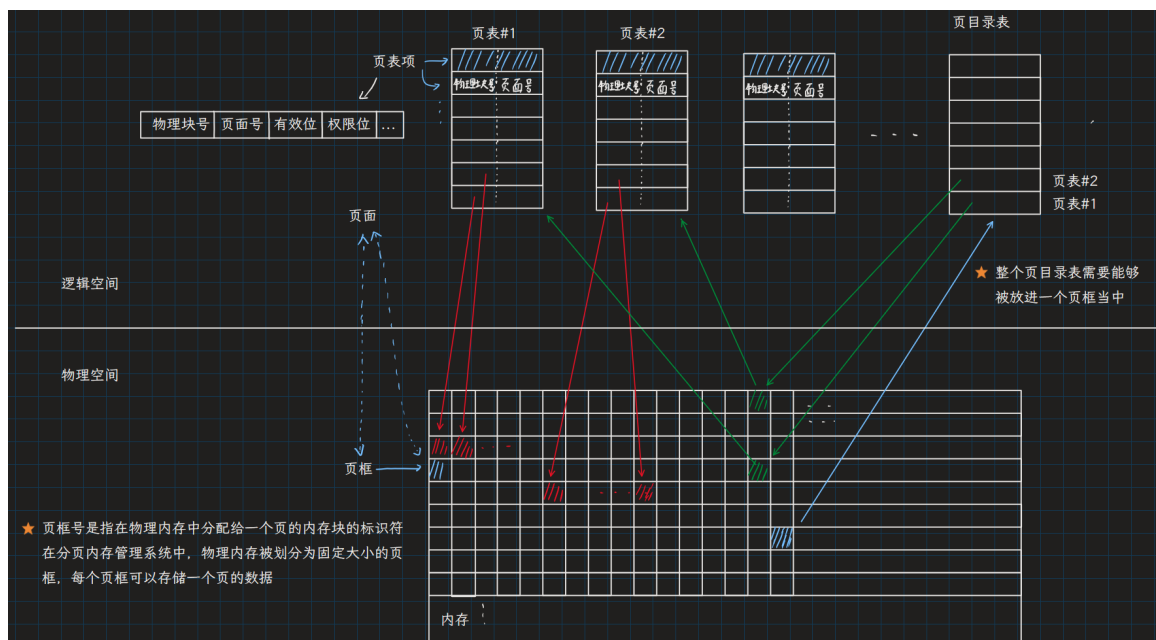
31		22	21		12	11		0
一级页号					二级页号		页内偏移量	
 - 两级页表结构：
 - 王道图示：



• 其他图示:



• 完整结构:



• 真题：2010年

• 解析：

【2010 统考真题】某计算机采用二级页表的分页存储管理方式，按字节编址，页大小为 $2^{10}B$ ，页表项大小为 $2B$ ，逻辑地址结构为

页目录号	页号	页内偏移量
------	----	-------

逻辑地址空间大小为 2^{16} 页，则表示整个逻辑地址空间的页目录表中包含表项的个数至少是 ()。

A. 64 B. 128 C. 256 D. 512

解：∵ 一页大小为 $2^{10}B$ ∴ 一张页表可以有 $2^{10}/2 = 2^9$ 张页表项
 ∴ 页目录需要 $2^{16}/2^9 = 2^7$ 个表项，才可表示整个地址空间。

• 总结：

- 根据页大小和页表项大小，可以知道**一张页表中可以存储多少个页面的信息**；
- 根据当前逻辑地址空间总大小，除以一张页表可存储的信息的大小，可以知道页目录表需要多少个表项，才能存储完整的地址信息；

注意：多级页表的大小以及访问次数

• 概念：

- 若采用多级页表机制，则**各级页表的大小不能超过一个页面**；

• 概念：两级页表的访问次数

- 第一次访存：访问内存中的页目录表；
- 第二次访存：访问内存中的二级页表；
- 第三次访存：访问目标内存单元；
- 多了一次访存，相比于单级页表性能消耗更大；

基于二级页表的逻辑地址转换

概念：地址转换

- 概念：
 - 为了方便实现地址变换，需要在系统中增设一个**外层页表寄存器** (也称页目录基址寄存器)，用于存放页目录始址；
 - 将逻辑地址中的**页目录号**作为页目录的索引，从中找到对应页表的始址；
 - 再用**二级页号**作为页表分页的索引，从中找到对应的页表；
 - 将**页表项**中的物理块号和页内偏移拼接，即为物理地址，再用该地址访问内存单元；
 - 以上共进行了 3 次访存；
 - 方法：
 - 1. 按照地址结构将逻辑地址拆分成三部分
 - 2. 从 **PCB** 中读出页目录表始址，再根据一级页号查页目录表，找到下一级页表在内存中的存放位置
 - 3. 根据二级页号查表，找到最终想访问的内存块号；
 - 4. 结合页内偏移量得到物理地址；
 - 举例：
 - 问题：
 - 将逻辑地址 (0000000000, 0000000001, 111111111111) 转换为物理地址 (用进制表示)；
 - 解析：
 - 访问页目录项：
 - 根据一级页表：0000000000，知道页表的索引号，即页面项为 **0#**，查页目录表页表所在的内存块号：3
- 内存块号

0	3
1	...
.....	
1023	...
- 访问页表：
 - 读出二级页表后，用二级页号读出内容：0000000001，查表，得知当前页面在 **4** 号内存块当中；
- 内存块号

0	2
1	4
.....	
1023	...
- 计算物理地址：
 - 访问 4 号内存块，计算出该内存块的起始地址为 $4 * 4096 = 16384$ ；
 - 页内偏移量为 111111111111；
 - 相加得到最终物理地址： $16384 + 1023 = 17407$

分段存储管理方式

分段存储基本概念

概念：什么是分段

- 概念：
 - 分段系统将用户进程的逻辑地址空间划分为大小不等的段；
 - 例如，用户进程由主程序段、两个子程序段、栈段和数据段组成，于是可以将这个进程划分为 5 段，每段从 0 开始编址，并分配一段连续的地址空间 (段内要求连续，段间不要求连续，进程的地址空间是二维的)；
 - 在页式系统中，逻辑地址的页号和页内偏移量对用户是透明的，但在分段系统中，段号和段内偏移量必须由用户显式提供，在高级程序设计语言中，这个工作由编译程序完成；
- 分配规则：
 - 分段：
 - 以**段为单位进行分配**，每个段在内存中占据连续空间，但各段之间可以**不相邻**；
 - 段号：
 - 又称之为段名；
 - **段号的位数决定了每个进程最多可以分几个段**；
 - 段内地址：
 - 又称之为段内偏移；
 - **段内地址的位数决定了每个段的最大长度是多少**；
- 图示：



- 特点
 - 1. 按逻辑功能模块划分；
 - 2. 用户编程更方便；
 - 3. 程序的可读性更高；

概念：分段系统的逻辑地址结构

- 概念：
 - 分段存储管理的逻辑地址结构由**段号与段内偏移量**两部分组成；
- 举例：
 - 如图，假设以字节编址，段号为 16 位，段内偏移量为 16 位；

- 因此一个进程最多有 2^{16} 段，最大段长为 $2^{16}B = 2^6KB = 64KB$

• 图示：

31		16	15		0
段号			段内地址		

分析：分段与分页管理的对比

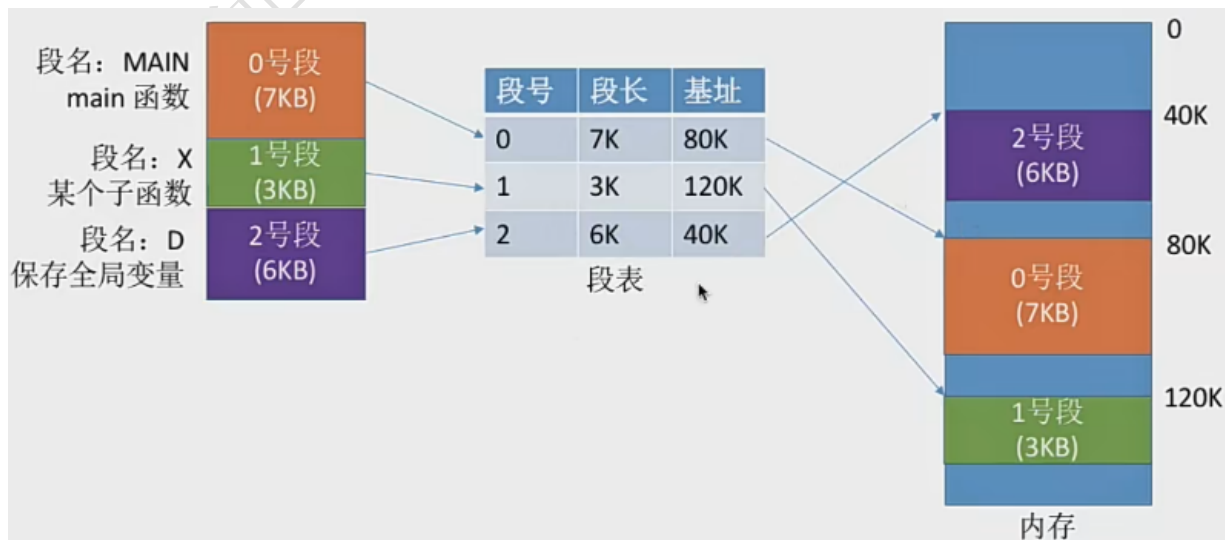
- (1) 页是信息的物理单位，分页的主要目的是提高内存利用率，分页完全是系统的行为，对分段的主要目的是更好地满足用户需求，用户用户是不可见的；
- (2) 段是信息的逻辑单位，按照逻辑关系将程序划分为若干段，分段对用户是可见的；
- (3) 页的大小固定且由系统决定。段的长度不固定，具体取决于用户所编写的程序；
- (4) 分页管理的地址空间是一维的。段式管理不能通过给出一个整数便确定对应的物理地址，因为每段的长度是不固定的，无法通过除法得出段号，无法通过求余得出段内偏移，所以一定要显式给出段号和段内偏移，因此分段管理的地址空间是二维的；

段表及段的共享

概念：段表

- 概念：
 - 每个进程都有一张逻辑空间与内存空间映射的段表，进程的每个段对应一个段表项，段表项记录了该段在内存中的始址和段的长度；
- 组成：
 - 1. 每个段对应一个**段表项**；
 - 2. 段表项中记录了该段在内存中的**起始位置**(又称基址)和**段的长度**；
 - 3. 各**段表项**的长度是相同的；
 - 4. 段号可以是隐含的，不占存储空间：若段表存放的起始地址为 M ，则 K 号段对应的段表项的存放的地址为 $M + K * \text{段表项长度}$

• 图示：



概念：分段与分页的共享

- 共享分析：

- 在分页系统中，虽然也能实现共享，但远不如分段系统来得方便；
- 若被共享的代码占 N 个页框，则每个进程的页表中都要建立 N 个页表项指向被共享的 N 个页框；
- 在分段系统中，不管该段有多大，都只需为该段设置一个段表项，因此非常容易实现共享；
- 不能被任何进程修改的代码称为可重入代码或纯代码（不属于临界资源），它是一种允许多个进程同时访问的代码；
- 为了防止程序在执行时修改共享代码，在每个进程中都必须配以局部数据区，将在执行过程中可能改变的部分复制到数据区，这样，进程就可对该数据区中的内容进行修改；
- **真题：**2010年
 - 若两个进程 P_1 和 P_2 同时共享段 S ；
 - 段共享时，内存中只保留一份段 S 的内容，**共享段表**中记录该段的物理地址和共享该段的进程数等相关信息；
 - 共享进程 P_1 和 P_2 的**段表**中各有一个**表项**指向该物理内存位置；
 - 二者各自对应的段表项所**指向**的物理空间位置相同，但逻辑地址不一定相同，即**段号是不一定相同**的；
 - 被共享的段，在共享段表中记录的共享该段的进程数为 0 时才回收该段的内存空间；

概念：段表的访问安全分析

- 与分页管理类似，分段管理的保护方法主要有两种：一种是存取控制保护，另一种是地址越界保护；
- 地址越界保护将段表寄存器中的段表长度与逻辑地址中的段号比较，若段号大于段表长度，则产生越界中断；
- 再将段表项中的段长和逻辑地址中的段内偏移进行比较，若段内偏移大于段长，也会产生越界中断。分页管理只需要判断页号是否越界，页内偏移是不可能越界的；

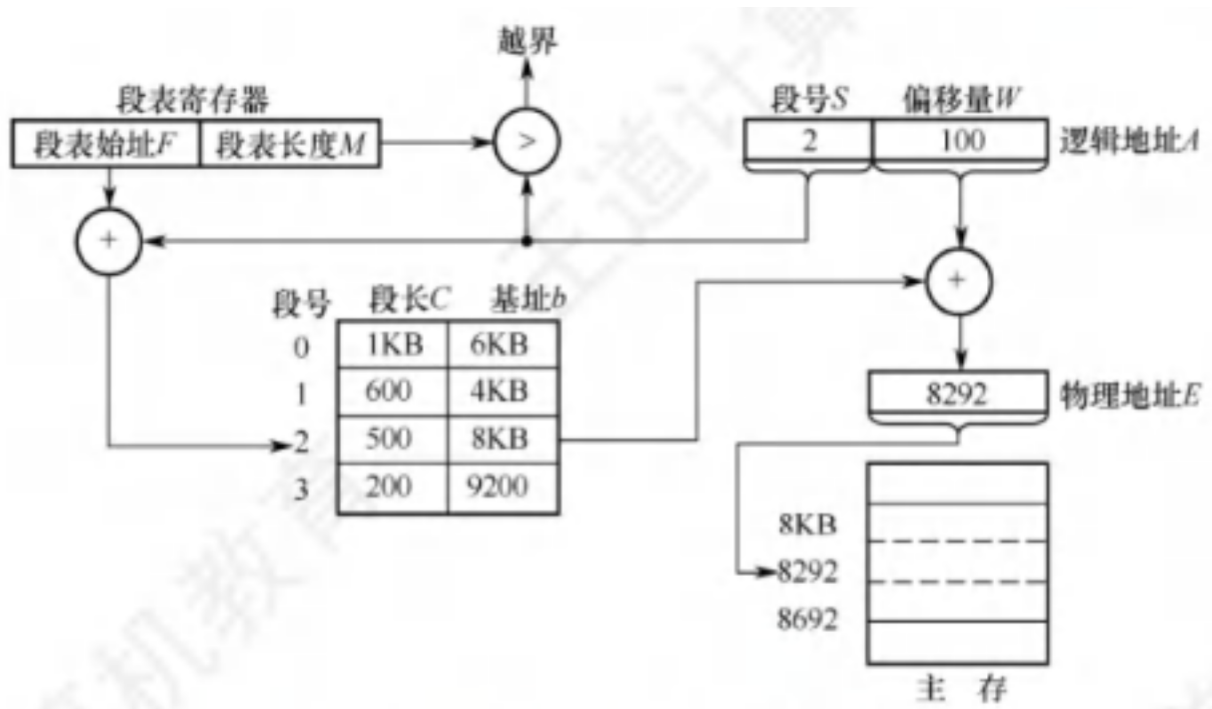
基于段表的逻辑地址转换

概念：段表寻址过程

- 过程：
 - 1. 根据逻辑地址得到：
 - (1) 段号；
 - (2) 段内地址；
 - 2. 判断段号是否越界，若 $S \geq M$ ，则产生越界中断，否则继续执行；
 - 注意：段表长度至少是 1 ，而段号从 0 开始；
 - 3. 查询段表，找到对应的段表项，段表项的存放地址为 $F + S * \text{段表项长度}$ ；
 - 4. 检查段内地址**是否超过段长**。若 $W \geq C$ ，则产生越界中断，否则继续执行；
 - 5. 计算得到物理地址
- 公式：
 - 物理地址计算：

$$\text{物理地址} = \text{段基址}b + \text{段内地址}W$$

• 图示：



• 真题：2016年

• 题目：

【2016 统考真题】某进程的段表内容如下所示。

段号	段长	内存起始地址	权限	状态
0	100	6000	只读	在内存
1	200	—	读写	不在内存
2	300	4000	读写	在内存

访问段号为 2、段内地址为 400 的逻辑地址时，进行地址转换的结果是（ ）。

A. 段缺失异常

B. 得到内存地址 4400

C. 越权异常

D. 越界异常

• 解析：

• 访问段号 2，得到其内存起始地址为 4000，并且段长为 300；

• 因为段内地址 400 > 段长 300，因此产生越界异常；

• 补充：越界异常和越权异常

• 越界异常 (Boundary Violation)

• 这种异常发生在程序试图访问其内存段之外的地址时。也就是说，程序试图访问的偏移量超出了该段分配的界限。

• 越界异常通常与段的界限检查有关。每个段都有自己的基址和界限，当访问的地址超出了这个段的界限时，就会触发这种异常。

• 例如，如果一个程序有一个从基址开始、长度为 1024 字节的代码段，而程序试图访问该段的第 1025 字节，就会发生越界异常

• 越权异常 (Privilege Violation)

• 这种异常发生在程序试图执行超出其权限级别的操作时。在许多操作系统中，特别是那些支持多级保护环（如 x86 架构）的系统中，不同级别的代码和数据具有不同的权限。

- 越权异常可能发生在程序试图访问更高权限级别的内存、执行特权指令或访问受保护的系统资源时。
- 例如，一个运行在用户模式下的程序试图执行一个只允许内核模式执行的指令，或者试图访问只读的系统文件，就会触发越权异常。

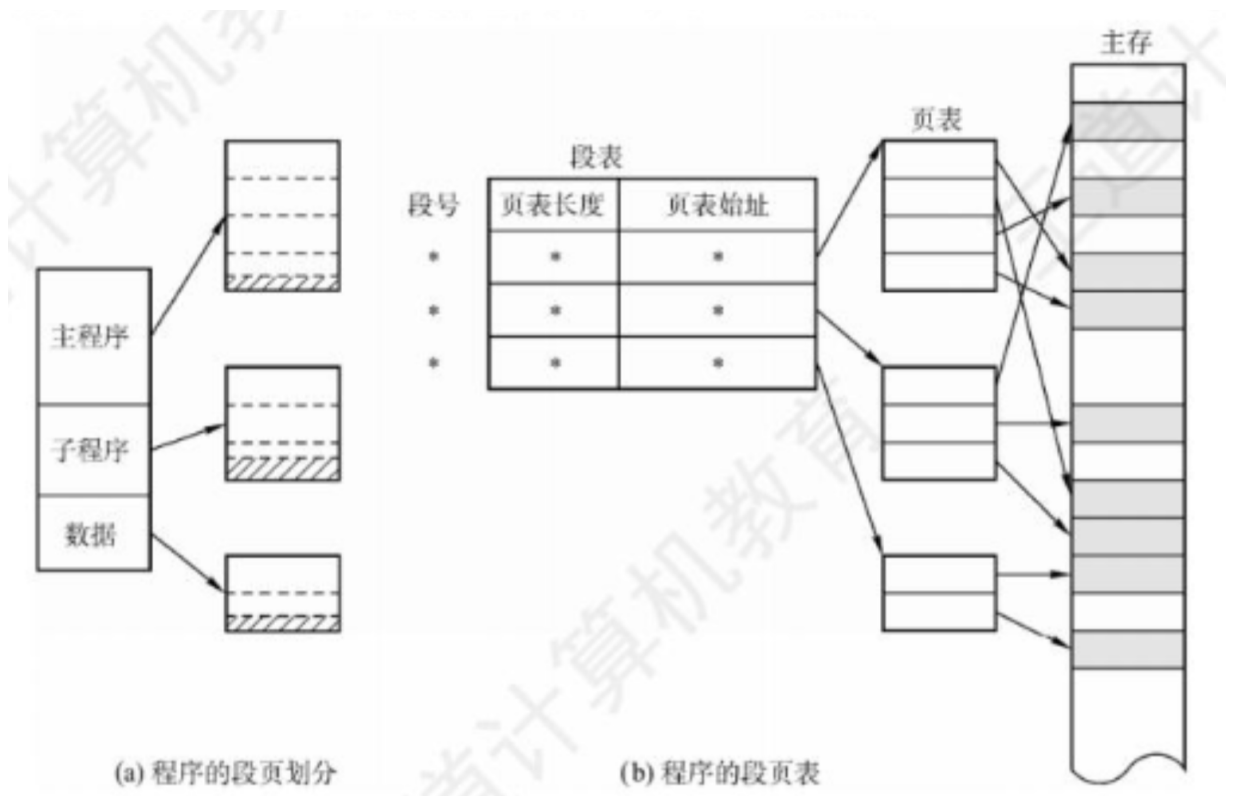
段页式管理方式

引入：分页与分段的优缺点分析

	优点	缺点
分页管理	内存空间利用率高， 不会产生外部碎片 ，只会有少量的页内碎片	不方便按照逻辑模块实现信息的共享和保护
分段管理	很方便按照逻辑模块实现信息的共享和保护	如果段长过大，为其分配很大的连续空间会很不方便。另外，段式管理 会产生外部碎片

概念：段页式管理方式

- 概念：
 - 在段页式系统中，进程的地址空间首先被分成若干逻辑段，每段都有自己的段号，然后将每段分成若干大小固定的页；
 - 对内存空间的管理仍然和分页存储管理一样，将其分成若干和页面大小相同的存储块，对内存的分配以存储块为单位
- 段表与页表：
 - 段表
 - 每个进程都有一张段表；
 - 每个段对应一个段表项；
 - 段表项长度相同，由段号（**隐含**），页表长度，页表存放地址组成；
 - 页表
 - 一个进程可以对应多个页表；
 - 每个页对应一个页表项；
 - 各页表项长度相同，由页号（**隐含**），页面存放的内存块号组成；
- 图示



• 图示:

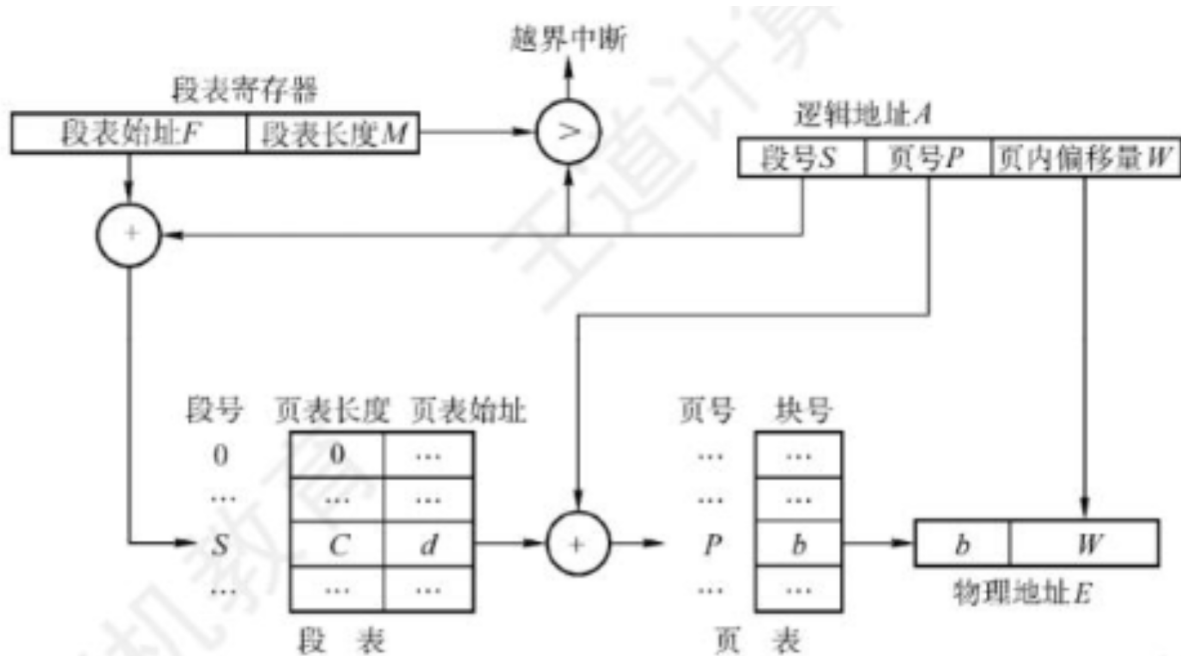
31		16	15		12	11		0
段号				页号		页内偏移量		

概念: 段页式管理的地址转换

• 过程:

1. 由逻辑地址得到段号, 页号, 页内偏移量
2. 段号与段表寄存器中的段长度比较, 检查是否越界
3. 由段表始址 + 段号找到对应段表项
4. 根据段表中记录的页表长度, 检查页号是否越界
5. 由段表中的页表地址, 页号得到查询页表, 找到相应页表项
6. 由页面存放的内存块号, 页内偏移量得到最终的物理地址
7. 访问目标单元

• 图示:



获取笔记持续更新可加微信: WandDZ