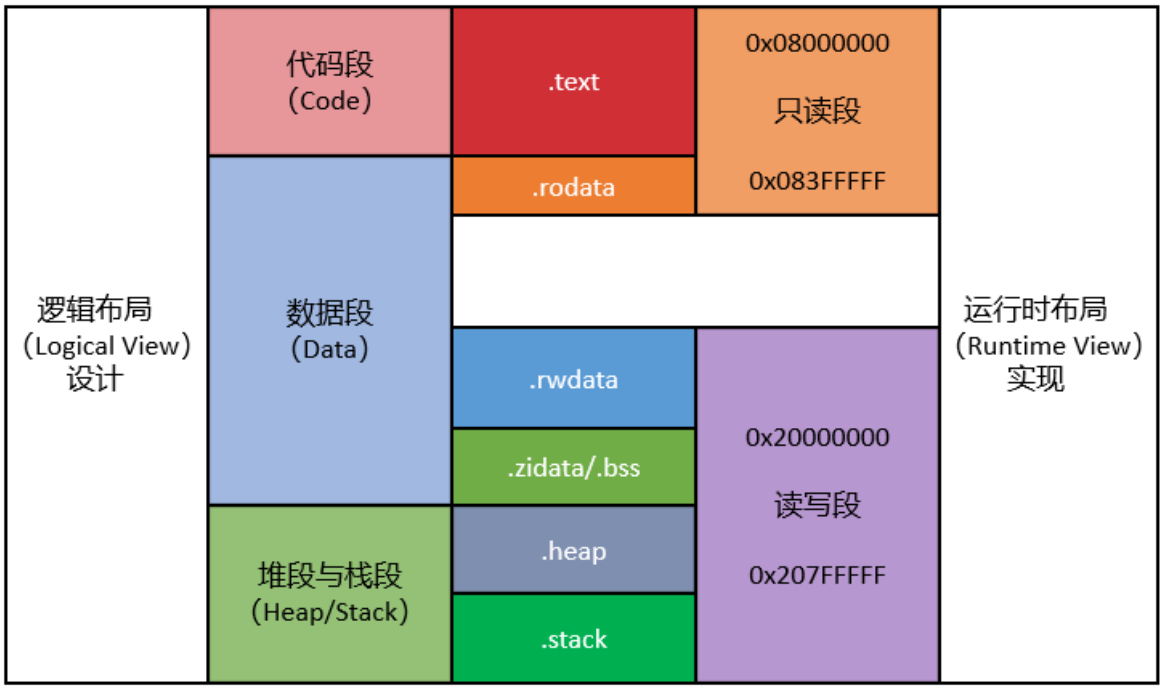
3.1 简单程序的结构

一.运行时视图

1.1代码段

运行时视图 程序在执行时的主存储器布局。也叫运行时布局。



代码段（.text） 存放程序的可执行指令，所有的执行都在代码段发生。

1.2数据段

数据段（.data）存放程序的数据，又可以细分成三类。

- **只读数据段（.rodata）**

存放程序中的含初值常量。这些常量在程序运行途中**不得修改**

- **读写数据段（.rwdata）**

存放程序中的含初值常量。这些常量在程序运行**可以修改。**

- **零初始化数据段（.zidata/.bss - Block Started by Symbol）**

存放程序中的不含初值（初始化为0的）可修改常量

1.3堆段与栈段

- **堆段（.heap）**

存放程序的堆，也即动态分配内存时内存的来源。

- **栈段（.stack）**

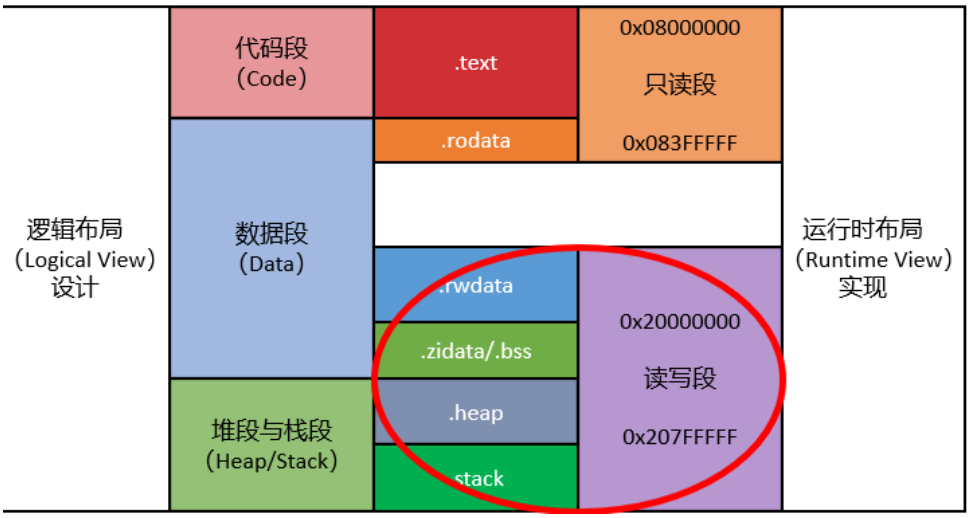
存放程序的运行栈，以供过程调用时保存和恢复上下文。

二.程序的装入

2.1代码的搬运

上电状态 与外存不同，内存在刚通电时内容是空的。

问题 至少哪些段需要放在内存里面？提示：要求可读写属性。



问题 那么，只读段呢？能从外存直接运行代码吗？

一般不行，除非外存是特制的，允许按字读取，也即可原位执行（XIP，eXecute-In-Place）。这在小型嵌入式设备中（Flash） 很常见。在这种场合，我们可以假设代码段已经到位了，由代码段去加载其它段。但如果代码段也没加载呢？

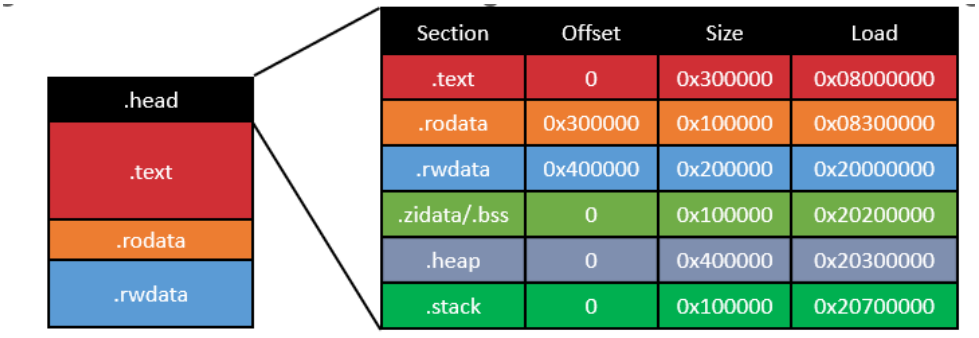
程序的装入 这意味着内存中现在**没有任何与这个程序相关的东西**。因此， 需要操作系统从外存读取程序文件的**逻辑段**，并按照其属性装入内存中。

外存上的程序 程序在外存上又是如何保存的呢？哪些段是必须保存的？除了 保存这些段的内容，还必须保存这些段的什么信息？

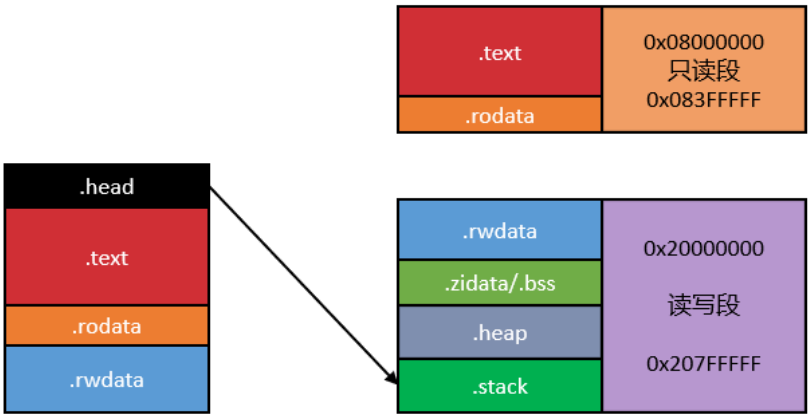
可执行文件 程序在外存上的储存方式，本质是保存了程序的逻辑布局的描述。

程序的装入 操作系统读取外存上的可执行文件中对程序逻辑布局的**描述** （Description），在内存中生成程序的物理布局的一个**实例** （Instance）的过程。

可执行文件头 .head 描述程序运行时布局的元数据，一般附加在可执行文件头部。



2.2数据的搬运



代码段

操作系统从可执行文件中读取代码段，并拷贝到内存中的指定地址。

操作系统在拷贝完成后设定该段为可读可执行段。

只读数据段

与代码段类似的拷贝过程。

可读写数据段

操作系统**从可执行文件中**读取可读写数据段，并拷贝到内存中的指定地址

零初始化数据段

操作系统将**内存**中的指定地址清零。

堆段

操作系统在**内存**中的指定地址初始化堆数据结构。

栈段

操作系统通常对该段什么也不做。

最后，在将控制权交给程序时，将**PC指向.text段，将SP指向.stack段**

三.程序的生成

3.1编程语言与工具链

编译器、汇编器、链接器、调试器与解释器程序

1. **编译器** 把**高级语言**源程序翻译成**机器语言**程序。

   有时也先翻译成汇编语言源程序，然后调用汇编器。

   如：GCC（C，C++等）、MSVC（C，C++等）。

2. **汇编器** 把**汇编语言**源程序翻译成**机器语言**程序。

   如：AS，MASM，TASM。

3. **链接器** 将**机器语言程序中间文件**与**系统运行库**链接生成**可执行的机器语言**程序。可能重定位各个段的位置。

   如：LD，LINK。

4. **调试器** 系统提供给用户的能监督和控制用户程序的一种工具，可以装入、 修改、显示或逐条执行一个程序。

   如：GDB，DEBUG。

5. **解释器 直接在机器上解释并执行高级语言源程序。**

   也不排斥使用编译器技术，内部先生成机器语言程序再执行。但是，解释器一般**不会生成可独立运行的机器语言程序文件。**

   如：Lua、JavaScript、Python。

工具链

编写、链接、调试应用程序往往需要一系列工具的帮助，这一 系列工具被称为工具链。

工具链 基本的编译工具链必须包括编译器、汇编器和链接器，否则无 法生成应用程序。若条件允许，还要包含调试器以便增进调试效率。

编译器

编译器 负责将输入的**高级**语言源程序翻译为**汇编**语言源程序。

高级源程序 程序员**手工编写的高级语言程序。**一个高级语言程序可以由多 个编译单元（**.C/.CPP**）组成，需要针对每个编译单元调用一次 编译器，分别生成对应于它们的汇编文件（**.S/.ASM**）。

当然，现代编译器都具备**直接生成机器码目标文件（*.O/*.OBJ）** 等的能力，可以跳过汇编文件这一步，只是原则上经过了汇编 语言这一层次。

清单列表文件 :**供程序员参考的一些可选信息**，诸如生成的汇编程序具体与高级语言怎样对应等等，甚至生成混合高级语言代码与汇编代码 的参考文件。这些信息对工具链是无用的，对程序员则很有用

汇编器

汇编器 负责将输入**源程序**中的指令进行**组装**，生成初步的机器码**目标文件。**有时，汇编器还生成供程序员参考的**清单列表文件。**

汇编源程序 程序员**手工编写的汇编语言程序。**一个汇编语言程序可以由多 个汇编单元（***.ASM**）组成，需要针对每个汇编单元调用一次汇 编器，分别生成其目标文件。

目标文件 汇编器组装指令得到的、包含初步机器码的文件（***.OBJ**）。这些 文件还不能直接被执行，需要进一步链接，确定程序中所含地 址的确切值后才可以直接被执行。

清单列表文件 供**程序员**参考的一些**可选信息**，诸如生成的机器码具体与汇编程序怎样对应、某些定义在第几行被定义，在第几行被引用等等。这些信息**对链接器无用**，完全是为程序员方便而引入的。

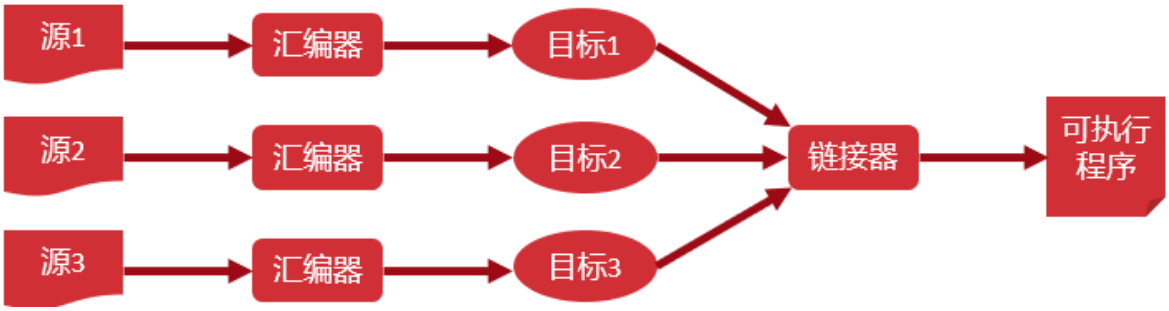
在MASM中，*.LST文件负责前者*，.CRF文件负责后者，但是只 有*.LST文件是文本文件*，.CRF则是需经转换才可读的文件。

链接器

链接器 负责确定程序各部分所对应的**具体地址**，并根据该地址**填充所有符号引用、生成最终可执行二进制文件**。有时，链接器还生 成供程序员参考的地址映射文件。

可执行文件 可直接在操作系统中执行的应用程序（***.EXE或.COM***），含有完全成熟的、可直接执行的机器码。

地址映射文件 供**程序员**参考的一些**可选地址映射信息**，诸如某段程序或数据最终被链接到哪个地址，等等。



3.2程序的编译与链接

目标文件的内容

外部符号引用

在一个目标文件中可能**引用了其它目标文件的内容**。由于编译器**一次生成一个目标文件**，因此并不知道那些引用的内容的地址在哪。况且，**自身在内存中的位置也还没确定**，因此所有对符号的引用（全局变量访问、函数调用等等）的具体地址都**只能推迟**。

extern关键字 表明引用的符号在当前的C或C++编译单元中未被定义，需要到**其它编译单元**中去寻找。在调用外部过程库时相当常用。

链接器的操作 链接器会先**收集全部目标文件的符号**，然后给每个符号**分配地址**。在地址确定后，**反过来补全**程序中对这些符号地址的引用。

直接回填法

直接回填 链接器生成所有符号的地址后，直接修改.text段中对这些符号的引用，将正确的地址填写到那些引用中去，这样生成的代码就 可以访问那些符号了。

直接回填法会**修改.text段**。

间接地址法

间接地址 编译器或汇编器生成目标文件时，在.rwdata段留出一个表格，.text段中的代码对其它符号的引用均通过这个表格进行。

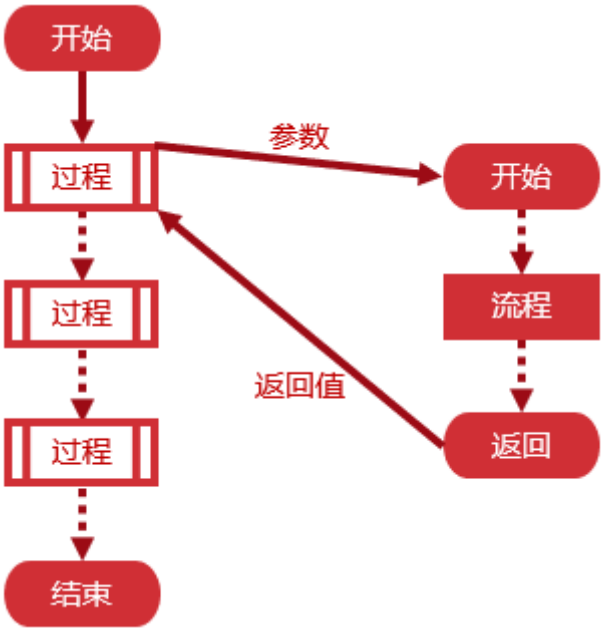
间接地址法不会修改.text段，但会**修改.rwdata段。**

3.3过程调用与栈框

过程（子程序）结构

程序指针可能在中途反复跳转到同一段过程执行，完毕后返回原处继续执行。

比循环结构更强大，因为过程的尾递归可以实现循环。过程还可以嵌套。



简单过程程序

近过程型调用者和被调用者处于同一代码段中，NEAR和PTR可以不写

远过程型调用者和被调用者处于不同的代码段内，FAR至少写一处

复杂过程程序

含参数型

调用者向被调用者传递一个或多个参数，参数传递有三种方法

（1)通过**全局变量**传递（需要在数据段声明）

（2）通过**栈**传递（记得设置SP和SS）

（3）通过**寄存器**传递（寄存器数量有限）

返回值型

被调用者向调用者返回一个或多个返回值

返回值的传递和参数一样，也有三种方法：（1)通过**全局变量**传递（需要在数据段声明）

（2）通过**栈**传递（记得设置SP和SS）

（3）通过**寄存器**传递（寄存器数量有限）

栈的四种类型

栈类型		递减满栈	递增满栈	递减空栈	递增空栈
增长方向		高→低	低→高	高→低	低→高
示意（压栈2次）		老SP 新SP	↑ 新SP 老SP	老SP 新SP ↓	↑ 新SP 老SP
栈指针位置		数据有效	数据有效	数据无效	数据无效
压栈	先	自减	自增	赋值	赋值
	后	赋值	赋值	自减	自增
弹栈	先	取值	取值	自增	自减
	后	自增	自减	取值	取值
代表架构		8086，RISC	8051，RISC	6502，RISC	RISC

传参约定

问题：参数和返回值都有多种方法，那么，调用一个函数时，具体采用哪种方法，以及哪种方法具体怎么实现？

传值约定be被调用者和调用者都遵循的变量和返回值的传递规则，具体怎么约定是随心所欲的。但是标准一旦确定下来，必须在整个项目中遵循，否则就会乱套。

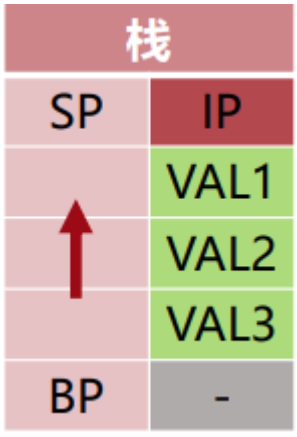
8086约定如下：

传递约定	STDCALL	CDECL	FASTCALL
中文名	标准调用	C声明调用	快速调用
参数	从后到前压栈	从后到前压栈	前两个放置在CX和DX中，其余从后到前压栈
返回值	AX	AX	AX
清理堆栈	被调用者	调用者	调用者
用途	Windows系统调用	C语言编译器的标准约定，汇编过程遵循它就能在C中被当成函数调用。	对速度有要求的场合

问题 为什么绝大多数调用约定都把参数放在栈上？为什么不使用寄存器传递所有的参数，或者使用变量传递所有的参数呢？

栈传参 栈的大小比较大，因此传递的参数可以比较多，准确地讲数量是不受限制的。此外，栈传递参数非常好理解，一般是把当前SP赋给BP，然后将参数列表从右往左依次入栈就可以了。被调用者通过BP指针就可以寻址各个参数，调用完成后要将SP指针恢复到原状也只需要将BP再赋给SP，非常方便。之所以8086的BP寄存器被指定隐含SS段寄存器，就是为了方便栈传参。

栈框指针 BP叫做栈框指针，因为通过它就能找到与该过程相关的整个调用栈，调用栈包括参数，临时变量，还要返回时的IP



问题 使用栈传递参数太麻烦了，要保存栈框，还要恢复栈指针。那么，在小型程序中，能否在数据段定义几个变量传递参数呢?

变量传参 传递参数也可以规定使用变量，比如某几个变量作为过程的输 入，某个变量作为过程的输出。**变量传参适合人类思维，因为参数的名称就是变量名，一目了然。**然而，使用固定变量传参的过程是不可重入的，也即一个过程的调用链中不能包含它自 己，因为变量正在被同一个过程的上一个未结束的调用使用。 这等于是说，不能使用任何形式的递归调用。 变量传参在那些程序存储器小、栈操作不便，且架构寄存器少的地方很有用，如8051等经典微控制器。这些架构的链接器会 使用覆盖分析（Overlay Analysis），判断哪些过程不在一条调用链上，并尽量将它们传参使用的变量复用以节约数据存储器。

问题 使用栈和变量传递参数非常慢，因为要访问内存。那么，能否彻底避免在传递参数时访问内存呢?

寄存器传参 传递参数还可以规定使用寄存器，比如某几个寄存器作为过程的输入，某个寄存器作为过程的输出。寄存器传参不需要访问存储器，因此速度很快，但是却需要消耗寄存器。**一旦寄存器用完了，多余的参数就只好通过栈传递了。**因此，在寄存器本来就少的CISC架构上，**寄存器传参往往仅限于前一两个参数。**

传参约定选择 一般程序，**推荐使用栈传递参数**；对速度要求不高的小程序， 以及新手上路，推荐使用**变量传递参数**，这样不容易错，当然 也可以**使用栈传递参数**；对性能要求极高的计算核，推荐使用寄存器（包括向量寄存器在内，它们空间很足）传递参数。

保存约定

问：每个被调用者都可能使用一些临时变量。这些临时变量可能在 三个地方：全局变量、栈、或者甚至寄存器中。如果这些变量 在栈中，那么分配参数时只要在栈上多分配一些空的位置（使 用SUB SP, imm）就可以了。但如果它们被分配在寄存器中，就 可能干扰调用者本身的执行，因为调用者可能仍然正在使用那 些寄存器。

保存约定

被调用者使用的寄存器，由调用者负责保存（压栈）还是由调用者负责保存，以及SP指针由调用者负责归位（弹栈），还是由被调用者负责归位。

调用约定 传值约定与保存约定合起来称为调用约定。

寄存器保护约定

调用者负责

被调用者假设所有的寄存器都可以随便使用。调用者负责保存它自己用到的寄存器。在下例中，被调用者使用AX~DX四个寄存器，而若调用者的AXDX中含有有用数据，则要自己保护。

优点 可以少保护寄存器。上例中被调用者SUBP（及其调用链）会使 用AX~DX四个寄存器，而调用者知道这一点，且它还知道自己调 用SUBP时AX和BX均不包含有效数据，那就只保护CX和DX即可， 即使AX与BX被修改也无妨。这样，代码的执行效率就比较高。

缺点 需要明确知道被调用者（及其下的调用链）会触碰哪些寄存器， 以及自己的哪些寄存器里面有有效数据，才能针对（哪怕是同 一个过程在不同地方的）不同调用实例生成最小的保护列表。 对人而言不那么容易，而且保护列表要反复写，增加代码量。

被调用者负责 调用者假设所有的、没用来进行参数传递的寄存器都保持不变，被调用者负责确保这一点。在下例中，被调用者使用AX~DX四个寄存器，而若调用者的CX和DX中含有有用数据，则要提前压栈。

优点 每个过程都负责清理自己的遗留，容易做到权责一致、做到模 块化，且保护列表只用写一次。如果连自己动过哪些寄存器都 懒得想，直接保护整个通用寄存器组（含FLAGS）完事。

缺点 这无疑会产生大量不必要的PUSH和POP，拖慢程序执行速度。

混合制负责 部分寄存器由调用者负责保存，另一部分寄存器则由被调用者 负责保存。下例中，AX、BX由调用者负责保存，CX、DX则由被 调用者负责保存，且AX~DX中均含有调用者的有效数据。

优点 一方面有利于生成尽量少的PUSH和POP，减少代码量，并方便被 调用者内部决定保护哪些寄存器（若被调用者也不使用那个寄 存器，则不管便可），另一方面也允许在每次调用时定制保护 列表，触碰尽量少的寄存器。

一方面允许编译器进行高度复杂的优化，另一方面也方便人手 写汇编。

缺点 约定复杂，写程序时要准确记得哪些寄存器是谁负责保存，一 旦弄错后果就很严重。

栈框恢复约定

调用者负责 被调用者不需要保证SP指针在RET结束后回到它被调用之前的状 态。SP指针的调整由调用者负责。

优点 可以最少化栈框调整。如果一个过程被连续调用两次，或者在 循环中被反复调用，或者尾递归，那么也许调整一次SP就足够了， 已分配的栈框本身可以重新填值并复用。另外，栈框怎么调整 是调用者说了算，因此调用者可以更灵活地处理每一次调用。 这在那些可变参数函数（如C语言printf）中非常有用。

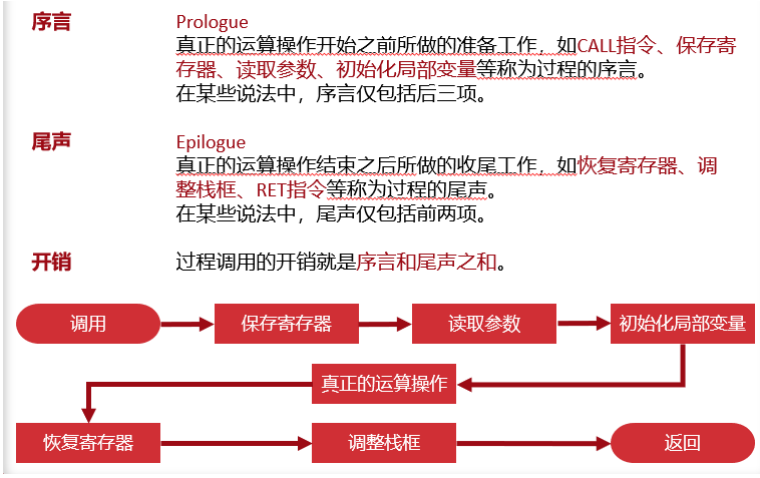
缺点 需要记住被调用者的栈框要怎么调整。对人而言这不太好记， 因此我们总是先保存SP到BP，然后结束调用后将BP赋给SP。

被调用者负责 被调用者需要保证SP指针在RET结束后回到整个调用序列之前的 状态。调用者无需调整SP指针。

优点 非常适合手写汇编，因为每个过程要弹多少字节的变量区和参 数区是写过程的时候就想明白的（算算用到多少个栈参数和栈 变量就可以）。在算一次之后，调用过程时候就再也不需要去 记要弹多少栈。另外，在那些支持SP相对寻址的i386机器上连BP 都可以解放出来（gcc -fomit-frame-pointer；可惜8086不行）。

缺点 刚性强，灵活性差，无法复用栈框，且难以处理不定参数的情 况。RET指令只能带立即数，因此只能使用JMP指令外加手动SP 调整来处理不定参数的情况。

过程执行的全景（STDCALL）



3.2 复杂程序的结构

一.多个工程共用

1.1语言运行时库

编程语言的库文件

比如c语言标准库：

头文件	作用	头文件	作用
stdio.h	输入输出	float.h	描述浮点数特性
string.h	字符串处理	limits.h	描述整形数特性
memory.h	内存管理	assert.h	诊断程序出错
math.h	数学函数	locale.h	程序的本地化
stddef.h	常用定义	setjmp.h	异常处理与返回
ctype.h	字符处理	signal.h	信号处理
stdlib.h	实用函数	time.h	日期和时间处理
errno.h	错误号定义	complex.h	复数处理

运行时环境

高级语言发展，要求**垃圾回收，虚拟机管理，即时编译**等功能，

所以，现在高级语言运行时实际上是一种环境，不仅仅只库本事

运行时环境运行砸用户态，他们会使用系统调用

1.垃圾回收

c语言中，内存要手动释放，Java等语言运行时环境会自动探测哪些对象已经被废弃并释放他们

2. 虚拟机管理

Java等语言的可执行文件是字节码，无法直接交给处理机执行，必须用虚拟机解析执行

3. 即时翻译

虚拟机字节码解析执行效率低，即时编译器可以将热点部分的字节码临场转译为原生二进制代码，在CPU上直接执行。

1.2静态链接库

静态链接库太多，用什么方法解决这个问题?

压缩文件

静态链接库——*.OBJ文件的简单集合。

他通常是将一堆.OBJ 文件的内容合并在一起成为一个文件，包括这些*.OBJ文件中包含的各个符号，以及各个符号的内容，

静态链接库的后缀名是*.A或 *.LIB 这样在引用运行时库或者第三方程序库时就可以直接引用这个链接库

存储压缩

除了将*.OBJ做简单合集之外*，.A文件往往还会使用一定的压缩 算法。这是因为现代库（尤其是wxWidgets、Qt等图形界面库） 的*.OBJ总量实在是太大了，单库动辄几十甚至几百MB，占用存 储空间实在太多，因此干脆像真正的压缩文件那样使用压缩算 法来保存它们。

用7z等解压缩软件来解压静态链接库。

细节暴露

常见的静态链接库将大量*.OBJ及其符号信息直接暴露给了用户。

但静态链接库内部并非所有符号都是需要暴露给用户的，还有 一部分符号其实是库内使用的，并不作为外部接口。（类似public 和 private)

部分连接

一种静态链接库技术，将静态链接库中的*.OBJ进行预链接，并 且除去（或混淆）所有的内部引用符号。

1. 符号除去，所有库代码编译乘一个二进制映像
2. 符号混淆，将所有的内部符号以及对其的引用全部修改为无意义的随机字符串。

二.多道程序共存

2.1简单分区

程序地址空间冲突——多个可执行文件连接到的空间可能是相互冲突的

简单分区——将物理内存分割成几个块，一个块运行一个应用程序，或者放置一个应用程序的某个段，各个应用程序的各个段在连接时就决定好要放置在什么地方。

简单分区的优点:

1. 配置容易

   不需要硬件，可以随意链接到任何位置，直接编写链接器脚本知道链接器做出此种链接行为即可，

2. 权限控制简单

   想要应用程序之间不能随意访问，每个程序能够合法访问的物理内存范围组成内存保护表，设计硬件，针对每一次内存访问进行合法性检查，放置访问越界。

内存保护单元（MPU）

简单分区布局的存储器权限控制工具，

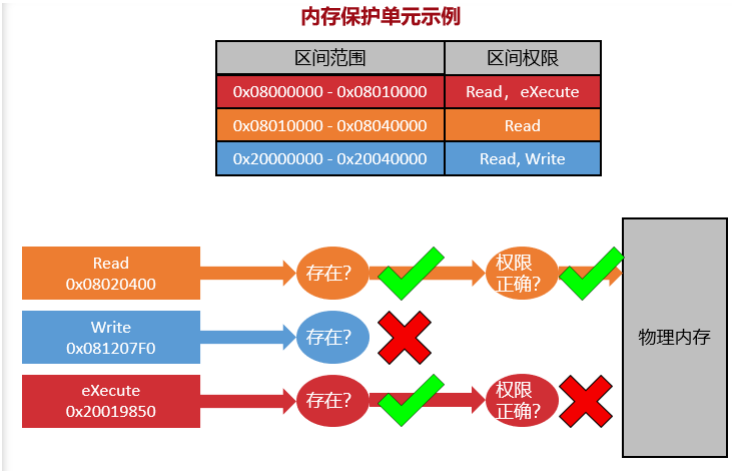
MPU使用一个内存保护表，表中有区间范围和区间权限一对信息。

每次访问内存都要经过内存保护表的权限检查：

1. 访问地址在某个分区的范围内
2. 访问的性质（读写执行）必须时该分区的权限允许的

价格低廉，实时性好。

例子：



简单分区的问题：

1. 二进制“球”

   很多应用程序厂商出于种种考虑（保护知识产权、减少程序体 积，等等）往往不会放出可二次链接的.*OBJ文件。它们往往会 放出*.BIN、.*HEX*、.EXE或其他格式的可执行文件，这些文件已 经采用直接回填法将其链接在了一个固定的地址，而且应用程 序的符号信息也都丢失了，无法进行二次链接。

   这种文件俗称“二进制球（ball/blob）”。它好像一个黑箱、一 个水晶球，拒绝外界修改，只能原样运行。

此时再发生地址冲突，怎么办？必须想办法重定位一些段。但 是软件的方法已经封死了。

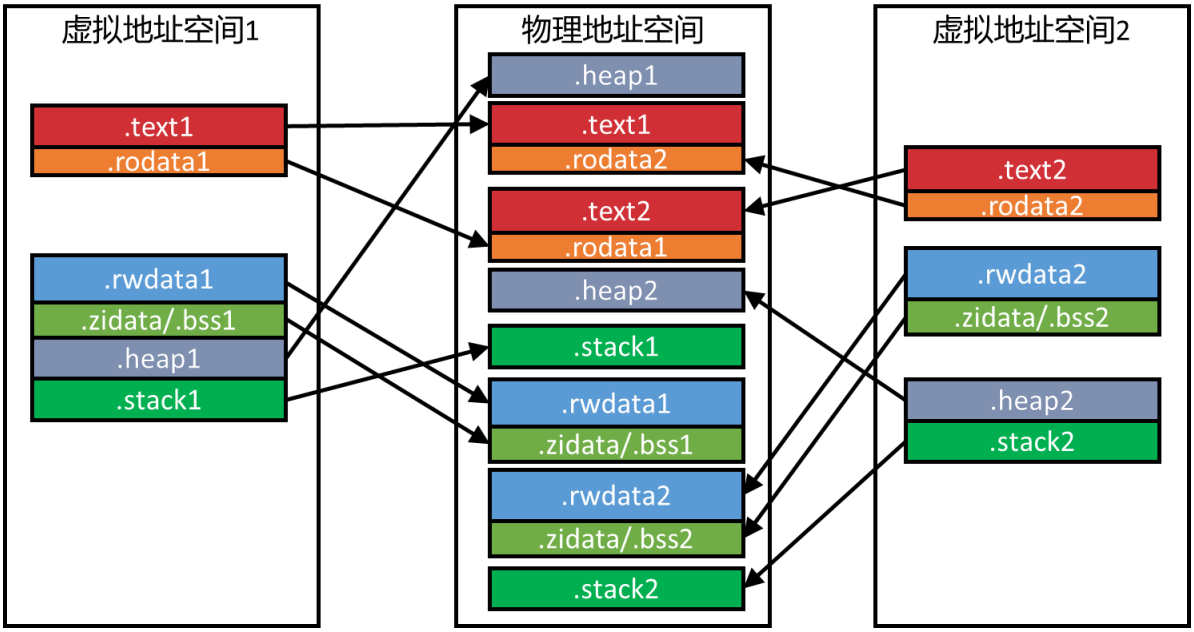
另外，向系统中动态添加程序困难重重。程序退出后，留下的空洞的地址和大小是不规则的，新的程序必须正好能适合这个空洞才行，这造成了很大困难。

因为这个原因，简单分区往往只适合那些应用程序大小固定、数量已知的场合。

2.2分段与分页

虚拟地址空间

硬件地址翻译 采用添加额外硬件的方法，将应用程序发起的存储器访问的地 址做系统性翻译，使得不同应用程序中对一个地址发起的访问 实际对应内存总线上的不同地址。



虚拟地址 应用程序认为它自己在访问的地址。也叫逻辑地址。

物理地址 CPU实际送出到内存总线上的物理存储单元地址。也叫实地址。

XX地址空间 由XX地址组成的地址集合就叫做XX地址空间。

内存管理单元 一种能依照某种规则将对逻辑地址的访问转换成对对应的物理 地址访问的硬件。（Memory Management Unit，MMU）

分段

按段划分虚拟地址 从程序的逻辑组织出发，其基本单元是一个个段。那么， 我们只需要将每个段重新映射到不同的物理地址就好了。 为此，我们需要给每个段分配一个段号，同时每个段都 对应一个物理地址区间，还拥有一个访问权限。

每个程序的虚拟段到物理段的映射关系组成段表。

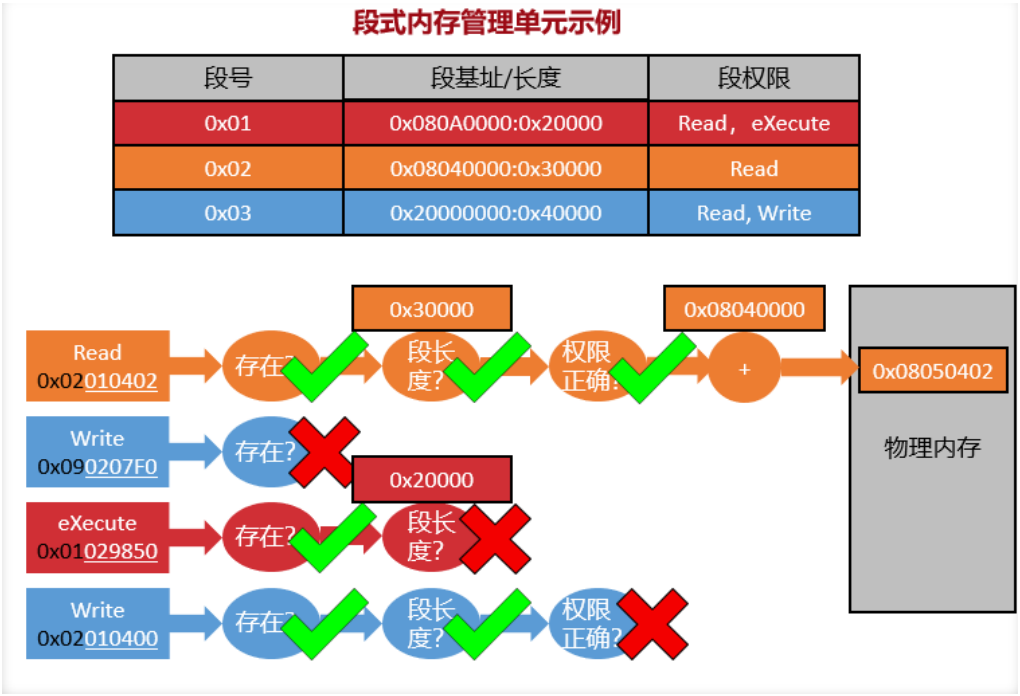
段式内存管理单元（S-MMU） 常用于分段布局的存储器访问管理工具，具备按段地址重映射和访问权限管理两个职能。段式内存管理单元使用一张段表，每个段都包括“段号”、“段物理地址范围”和“段权限”三个部分。由应用程序发起的每一次内存访问都需要经过段表指定的转换和检查：

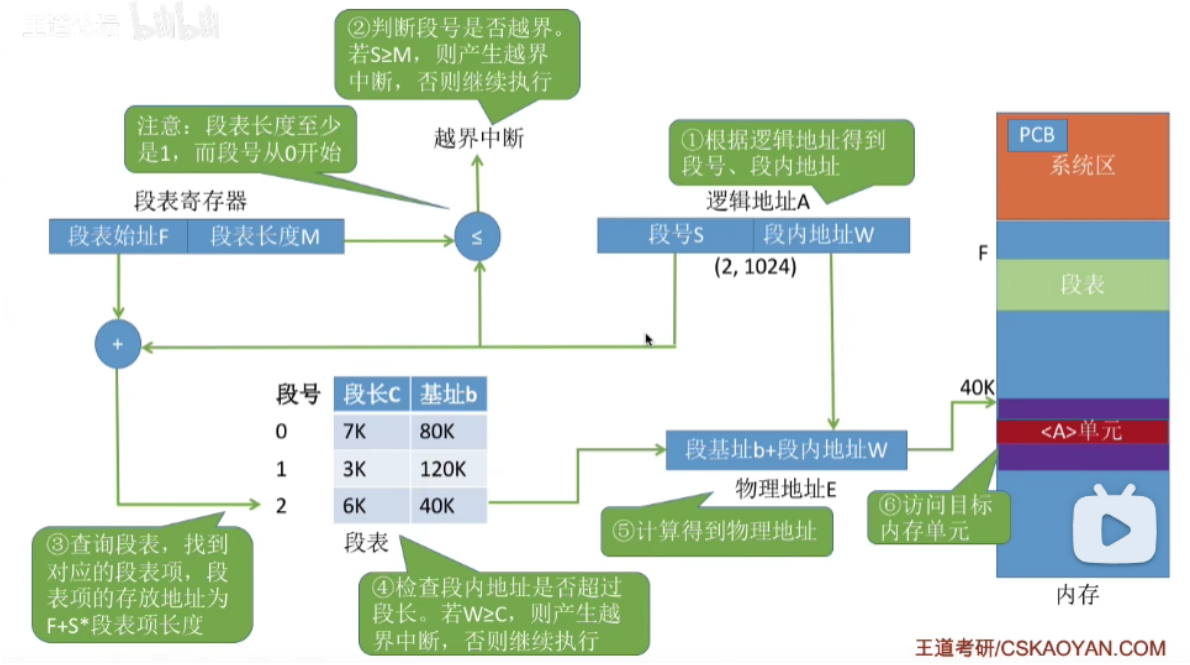
（1）按照访问的虚拟地址中的段号信息查找相应的段。

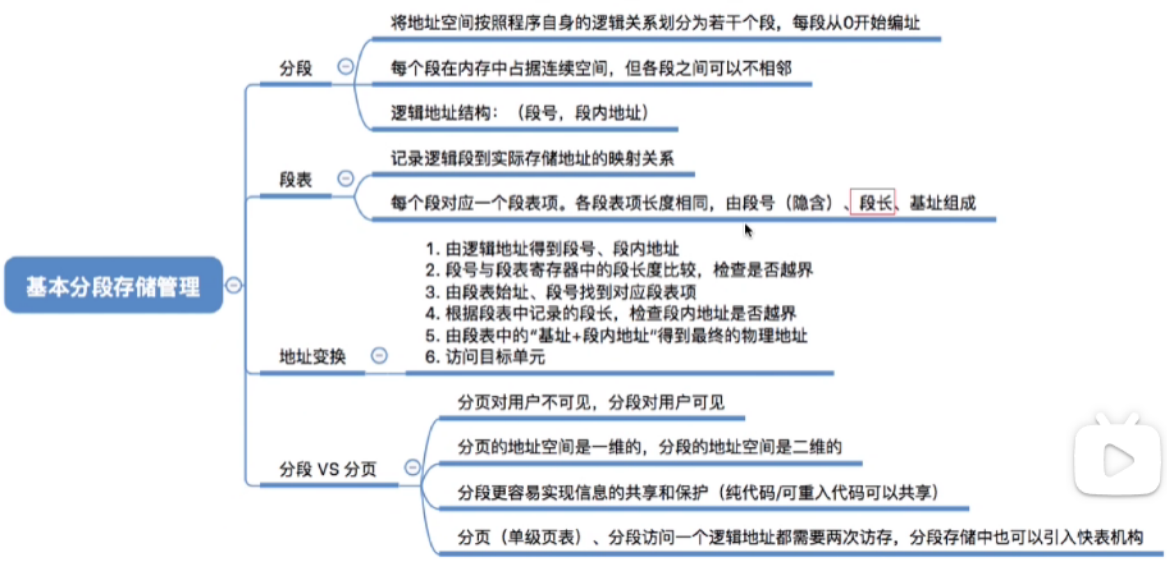
（2）发起的访问的性质（读，写，执行）必须是该段 的权限允许的。

（3）访问的物理地址=段基址+虚拟地址，且虚拟地址 不得超过段长度。

此种方法在硬件上仅需要一组比较器和一组加法器电路 就可以实现，是需要虚拟地址空间时的一种讨巧方法


段式内存管理单元示例
段号
段基址/长度
段权限
0x01
0x080A0000:0x20000
Read，eXecute
0x02
0x08040000:0x30000
Read
0x03
0x20000000:0x40000
Read, Write
Read
0x02010402
存在?
0x30000
段长度?
权限正确?
0x08040000
+
0x08050402
物理内存
Write
0x090207F0
存在?
0x20000
eXecute
0x01029850
存在?
段长度?
Write
0x02010400
存在?
段长度?
权限正确?


②判断段号是否越界。若S≥M，则产生越界中断，否则继续执行
越界中断
注意：段表长度至少是1，而段号从0开始
①根据逻辑地址得到段号、段内地址
段表寄存器
逻辑地址A
段表始址F
段表长度M
≤
段号S
段内地址W
(2, 1024)
PCB
系统区
F
段表
+
段号
段长C
基址b
0
7K
80K
1
3K
120K
2
6K
40K
40K
段基址b+段内地址W
<A>单元
物理地址E
⑤计算得到物理地址
⑥访问目标内存单元
③查询段表，找到对应的段表项，段表项的存放地址为F+S*段表项长度
段表
④检查段内地址是否超过段长。若W≥C，则产生越界中断，否则继续执行
内存
王道考研/CSKAOYAN.COM


基本分段存储管理
分段
将地址空间按照程序自身的逻辑关系划分为若干个段，每段从0开始编址
每个段在内存中占据连续空间，但各段之间可以不相邻
逻辑地址结构：（段号，段内地址）
段表
记录逻辑段到实际存储地址的映射关系
每个段对应一个段表项。各段表项长度相同，由段号（隐含）、段长、基址组成
地址变换
1. 由逻辑地址得到段号、段内地址
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. 根据段表中记录的段长，检查段内地址是否越界
5. 由段表中的“基址+段内地址”得到最终的物理地址
6. 访问目标单元
分段 VS 分页
分页对用户不可见，分段对用户可见
分页的地址空间是一维的，分段的地址空间是二维的
分段更容易实现信息的共享和保护（纯代码/可重入代码可以共享）
分页（单级页表）、分段访问一个逻辑地址都需要两次访存，分段存储中也可以引入快表机构

分页

按页划分虚拟地址 从物理地址空间的细粒度划分出发，我们将物理地址切 割成一个个大小相等的小页，并将虚拟地址也切割成同 样大小的页。那么，我们只需要将每个虚拟页映射到不 同的物理页就好了。为此，我们需要给每个页分配一个 页号，同时每个虚拟页都对应一个物理页，还拥有一个 访问权限。

每个程序的虚拟页到物理页的映射关系组成页表。

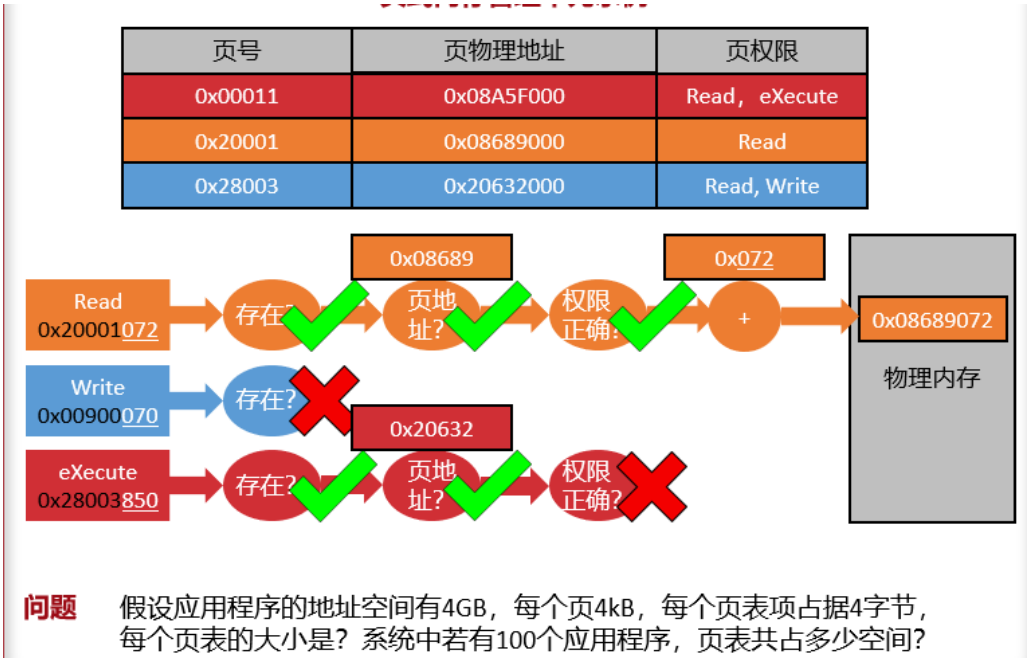
页式内存管理单元（P-MMU) 常用于分段布局的存储器访问管理工具，具备按页地址重映射和访问权限管理两个职能。页式内存管理单元使用一张页表，每个页都包括“页号”、“页物理地址”和“页权限”三个部分。由应用程序发起的每一次内存访问都需要经过页表指定的转换和检查：

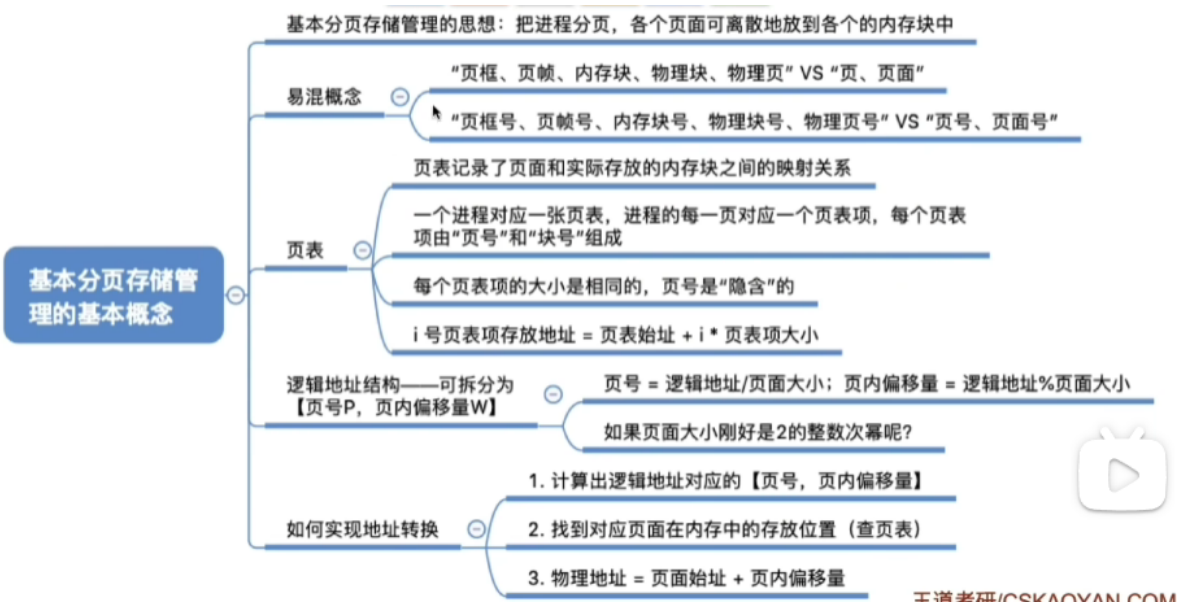
（1）按照访问的虚拟地址中的页号信息查找相应的页。

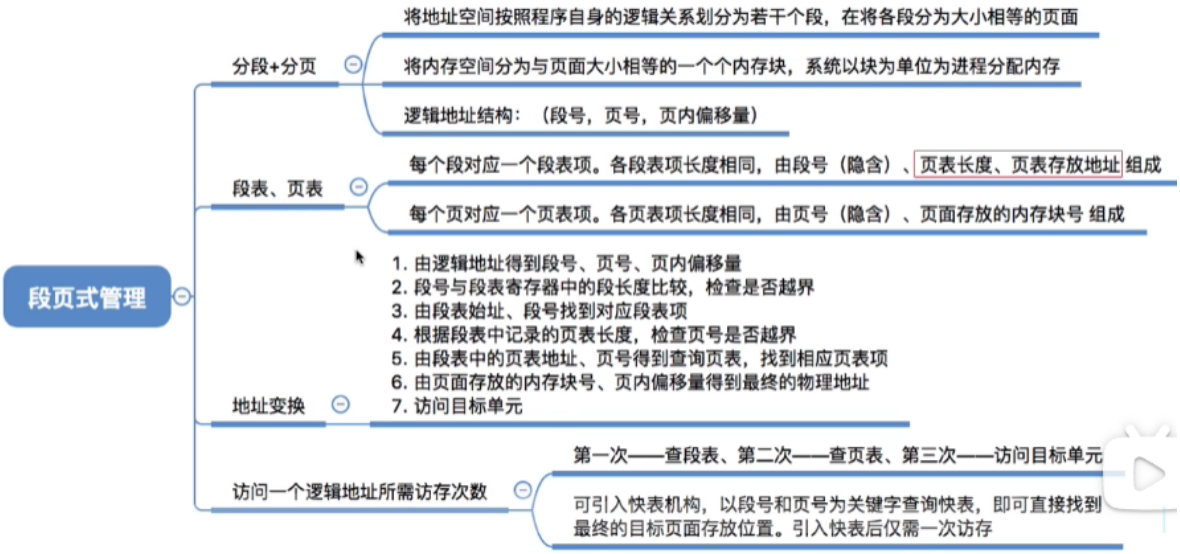
（2）发起的访问的性质（读，写，执行）必须是该页 的权限允许的。

（3）访问的物理地址=页基址+页内偏移量。

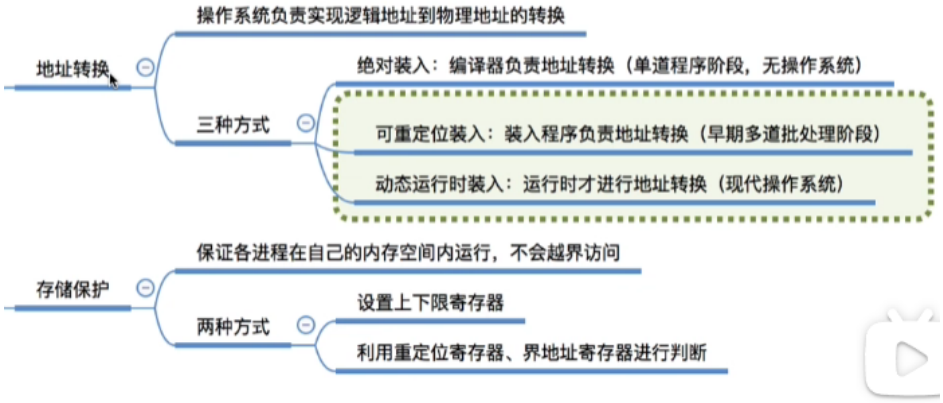
乍看之下比S-MMU还简单。但线性页表中页的数量...







三.多道程序共享



3.1动态链接库

静态链接的问题：

1. 重复存储，库会被每个可执行文件中保留一个副本，浪费磁盘
2. 更新困难，每次更新库后，都需要把用到它的可选择文件重新连接
3. 重复加载，库会被在每一个虚拟地址空间保留一个副本，进而导致在物理地址空间也产生副本，浪费内存，浪费载入时间

动态链接库

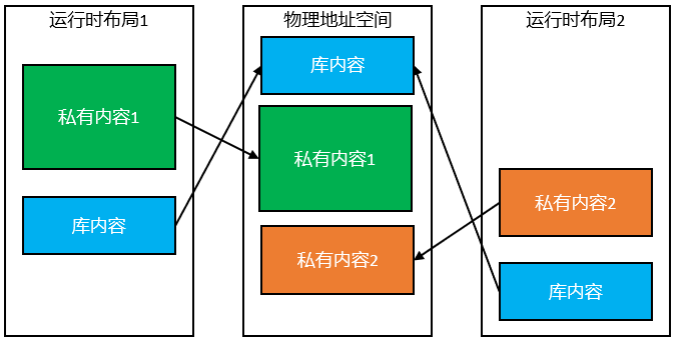
在可执行程序被加载时，作为独立文件单独加载的库文件。它不包含在可执行文件之内；可执行文件将在需要它们的时候加载它们。

动态链接库的后缀名一般是.*SO（Shared Object）*、.DYLIB （Dynamic Library）或*.DLL（Dynamic-Link Library）。

外存的库共享

可执行文件中不再包含库文件。这样，库文件就只要在外存中存在 一份。这个特征是动态链接库的根本特征；判断一个库是不是动态链接库，就看调用它的可执行文件中是否包含它的内容。如果不包含，它就是动态链接库。

内存的库共享 物理地址空间中仅包含库文件（的代码段）的一个副本。这样，即 便库文件被多个虚拟地址空间中被使用，它也只会占用一份物理内 存，减少了内存用量，也减轻了缓存负担。



问题：现在又静态链接库，能不能直接作为动态链接库来使用?

解决方案：将静态链接器做成静态链接库然后静态连接到应用程序，应用程序在启动时先启动静态链接器对自己和库进行静态链接再运行

优缺点：

优点

（1）静态链接的工具链和库可以直接用。

（2）无需操作系统的加载器支持动态链接。

（3）采用直接回填法时不产生任何性能损失。

（4）用不到的符号可以不链接到运行时视图。

缺点

（1）需要修改自己的各个段以添加静态库中的符号。

（2）如果采取直接回填法，还需要填充自己.text段里面的引用，速 度很慢，程序启动需要很长时间。

（3）不管这些库文件在执行中是否真的用到了，程序运行前必须 一次性链接程序声明引用的库，加剧了（2）。

（4）可执行文件中必须保留自己的全部链接信息。有可能被友商 或者黑客拿去反向工程。

（5）外存库可以共享，但加载到内存之后还是会产生多个副本

3.2外存的库共享

接口的定义：

应用程序和库之间是通过某些定义好的接口互相引用的。

原则上应用程序只要暴漏这些接口就可以了，库也一样。

这样没必要暴露多余的任何信息，在加载应用程序时也不需要动态链接这些信息

动态库的生成

在生成动态链接库时，将动态链接库中的符号都连接到某个固定虚拟地址，然后将动态链接库中被导出的接口符号以外的符号都出去。

这样动态链接库中的一起符号的位置都固定了，堆自己内部函数的一弄不再需要链接

在加载动态链接库时，只要将它加载到那个预链接的固定地址就好，然后修改应用程序中对他的引用，指向动态链接库即可。

链接器共享

前面的解决方案里面，链接器在每个可执行程序中都有一个副本。 这实际上没必要，如果操作系统能够提供一个链接器并且在加载可 执行文件的时候自动加载，就不需要把它放在可执行文件里面了

动态链接器

加载动态链接库并将其与应用程序相链接的链接器。它在可执行文 件加载时或应用程序运行时才运行。（静态链接器：在生成可执行 文件时运行）

发生虚拟地址冲突时：

动态库重定位

在生成动态链接库时，包含一些重定位信息，告诉动态链接器，如 果发生地址冲突，要修改动态链接库的什么地方来处理冲突

3.3内存的库共享

重定位的问题：

将动态链接库重定位后，如果需要多个应用程序共享一份内存副本 的话， 不管这个库重定位到哪里，多个应用程序都必须以同样的 虚拟地址引用这个库。

如果... 新加载进来的应用程序与已经加载好的动态库冲突怎么办?

解决方案一 再加载一个新的副本，重定位到与应用程序3的本体不冲突的虚拟 地址去。问题：物理内存中有两个库副本了。比三个副本还是来得 好，但是没达到目的。

解决方案二 规定动态链接库使用的地址范围为0x0000-0x7FFF，而应用程序只能 使用0x8000-0xFFFF，这样不可能出现应用程序和动态链接库冲突的 情况。但这限制了应用程序的布局，如果应用程序需要使用低地址 呢？而且，每一个动态链接库都必须占据一个独立的虚拟地址。

位置无关代码

加载到任何绝对地址都可以直接运行的代码。其所有的地址访问 （包括代码和变量）都相对于某指针，常见的是IP、SP或BP

改进后，不会再出现任何冲突问题，因为同样的一段代码不管映射到什么虚拟地址都工作，任何动态库只在物理内存中加载一次就可以了，不需要关心虚拟地址冲突问题，因为我们压根就没动库的代码段，库的代码段都是相对寻址。

库的数据段在内存中的共享

.rodata 本质上讲，.rodata和.text是一回事，只是不能执行罢了。它是只读的，共享它没有问题。

.rwdata 这里包含可读写的数据，不同的应用程序可能会给库传递不同的参数，因 此其实它们在逻辑上是独立的，不能共享。

.zidata 和.rwdata的情况是一样的，不能共享。

.heap 每个应用程序当然会以不同的方式使用堆。不能共享。这个段在库文件中 一般是不存在的，因为应用程序才有堆。库是没有自己独立的堆的。

.stack 和.heap的情况是一样的，不能共享。这个段一般也不存在，因为栈就是应用程序的栈。

怎么处理这些不能共享的段?

解决方案 考虑虚拟地址空间，只要将库的可读写段映射到不同的物理地址就 可以了，它们自然就分开了。当然，在虚拟地址空间，各个应用程 序加载的库文件访问的地址还是那个地址。

库的只读段则映射到同样的物理地址，避免产生不必要的拷贝。

动态链接库调用动态链接库

链接库互相调用 如果一个动态链接库要调用另一个动态链接库，怎么办?

注意事项 动态链接库的只读段在多个应用程序之间共享，不可更改， 无法做直接回填法。因为另一个动态链接库的加载地址在 不同虚拟地址空间之间可能也是不一样的。

思路 利用读写段多副本的特性，使用间接地址法，将自己引用的所有符 号的地址都放在可读写段的某个表格中，每一次函数调用和全局变 量存取时都先查询这个表格得到该符号在本虚拟地址空间中的地址，然后再访问那个地址即可。

当然，调用内部的符号时只要使用IP，SP或BP相对寻址就可以了。

3.4惰性动态链接

问题 随着应用程序的发展，一个程序引用的动态库越来越多，这包括它 们的语言运行时库、各种第三方库，甚至还有第三方库引用的第三 方库引用的第三方库（！）。

这些库之中有很多根本不会使用，可是为了方便编程但凡是个应用 程序就会声明自己可能引用它们。如果在应用程序启动时就加载或 者映射这些库，机器就要卡死。那么，能否将动态链接库的加载和链接推迟到真正使用它们的一刻呢?

推迟链接知道需要之时

直接回填法 直接回填法哪怕对应用程序都做不到，因为我们不可能让应用程序空着地址去跑。一旦跑到那条没填充的指令就干脆出错了。

间接填充法 也有问题，因为库还没加载，那些间接地址填充什么才好呢?填充0的话，又会出相同的问题了。

延迟绑定 在调用到某某符号或功能时才去查找和加载它。如果该符号或功能 一直不被调用，那么它们就永远不会被加载。

惰性动态链接 对于全局变量（一般而言不多），我们在程序启动之时就向间接跳 转表填充它们的地址。对于函数（动态链接库的主要成分）则不同， 我们可以在间接地址表里面填充一段专门用来查找函数真实地址的 代码。

这样，第一次函数调用将会跳转到查找代码，由查找代码来查找函 数的真实地址，再回填到间接地址表里面，最后再调用那个真实函 数。下一次访问间接地址表进行函数调用的时候，就会直接调用到 真实函数。

惰性动态加载 没有必要在程序启动时就映射整个动态库，仅加载其间接地址表就 足够了。当程序访问间接地址表指向的地址，发现那里没有加载 （映射）动态库，就会抛出异常，操作系统截获这个异常，在异常 处理中再加载动态库的对应部分就可以了。

改进效果 加载进来的仅仅是间接地址表，而且就连它也只加载了全局变量的 地址，函数地址则未加载。其它一切都是用到了才会加载的。

链接方式	复杂程度	细节隐藏	共享程度			加载效率	运行效率	实时性	安全性	版本管理
			工程	外存	内存					
目标文件直接链接	低	否	否	否	否	中	高	高	高	易
静态链接	低	否	是	否	否	中	高	高	高	易
提前静态链接	中	是	是	否	否	中	中	高	高	易
回填动态链接	低	否	是	是	否	低	高	高	中	难
重定位动态链接	中	是	是	是	部分	低	高	高	中	难
位置无关动态链接	高	是	是	是	是	中	中	高	低	难
惰性动态链接	高	是	是	是	是	高	低	低	低	难