

**1. 寄存器 (Register)**

* **位置**：CPU 内部
* **特点**：
  + 存取速度最快，与 CPU 的指令执行紧密结合。
  + 容量非常小，但每次存取时间几乎为零。
  + 主要用于临时保存指令或数据，如运算操作的结果。
* **示例**：在 CPU 计算时，寄存器存储即将参与运算的数据，如指令计数器（Program Counter）、累加器（Accumulator）。

**2. 高速缓存 (Cache)**

* **位置**：CPU 和主存之间
* **特点**：
  + **提高访问速度**：缓存常用数据，减少访问主存的次数。
  + **多级缓存**：现代系统中常见 L1、L2、L3 三级缓存，级别越高距离 CPU 越远，存取时间也越长。
* **意义**：
  + 避免频繁访问较慢的主存（RAM）。
  + **示例**：某个程序频繁调用相同的变量时，这些数据可能被缓存存储，提高响应速度。

**3. 主存储器 (RAM)**

* **位置**：主机内存条
* **特点**：
  + **动态随机存取存储器 (DRAM)**：常用于 PC 内存。
  + **静态随机存取存储器 (SRAM)**：更快但更昂贵，常用作缓存。
  + **速度** 远低于寄存器和缓存，但容量更大。
* **用途**：用于存储程序的运行数据和代码。

**4. 磁盘缓存 (Disk Cache)**

* **位置**：主存和外部存储之间
* **特点**：
  + 将一部分磁盘数据暂时缓存在内存中，提高存取速度。
  + **举例**：某个应用程序反复读取一个文件的不同部分时，磁盘缓存可以提前加载数据，减少 I/O 操作。

**5. 外存储器 (Secondary Storage)**

* **设备**：如硬盘、固态硬盘、光盘等。
* **特点**：
  + **容量大，速度慢**：主要用于持久化存储。
  + **数据持久化**：即使系统断电，数据依然保留。
* **意义**：用于存储不常用的数据，以及系统和用户的文件。

**总结：存储层次结构的设计意义**

1. **存储与速度的平衡**：越靠近 CPU 的存储器速度越快，但成本越高，容量越小。
2. **减少访问延迟**：通过多级缓存系统，尽可能减少 CPU 等待时间，提高处理效率。
3. **节省成本**：大容量数据存储在价格较低的外存储器中，但使用缓存和内存加速访问。

**代码示例：利用缓存提高访问效率**

c

复制代码

#include <stdio.h>

#define ARRAY\_SIZE 100000

int main() {

int array[ARRAY\_SIZE];

// 初始化数组

for (int i = 0; i < ARRAY\_SIZE; i++) {

array[i] = i;

}

// 多次访问同一块数据，模拟缓存优势

long sum = 0;

for (int j = 0; j < 1000; j++) {

for (int i = 0; i < ARRAY\_SIZE; i += 64) { // 模拟缓存行大小

sum += array[i];

}

}

printf("Sum: %ld\n", sum);

return 0;

}

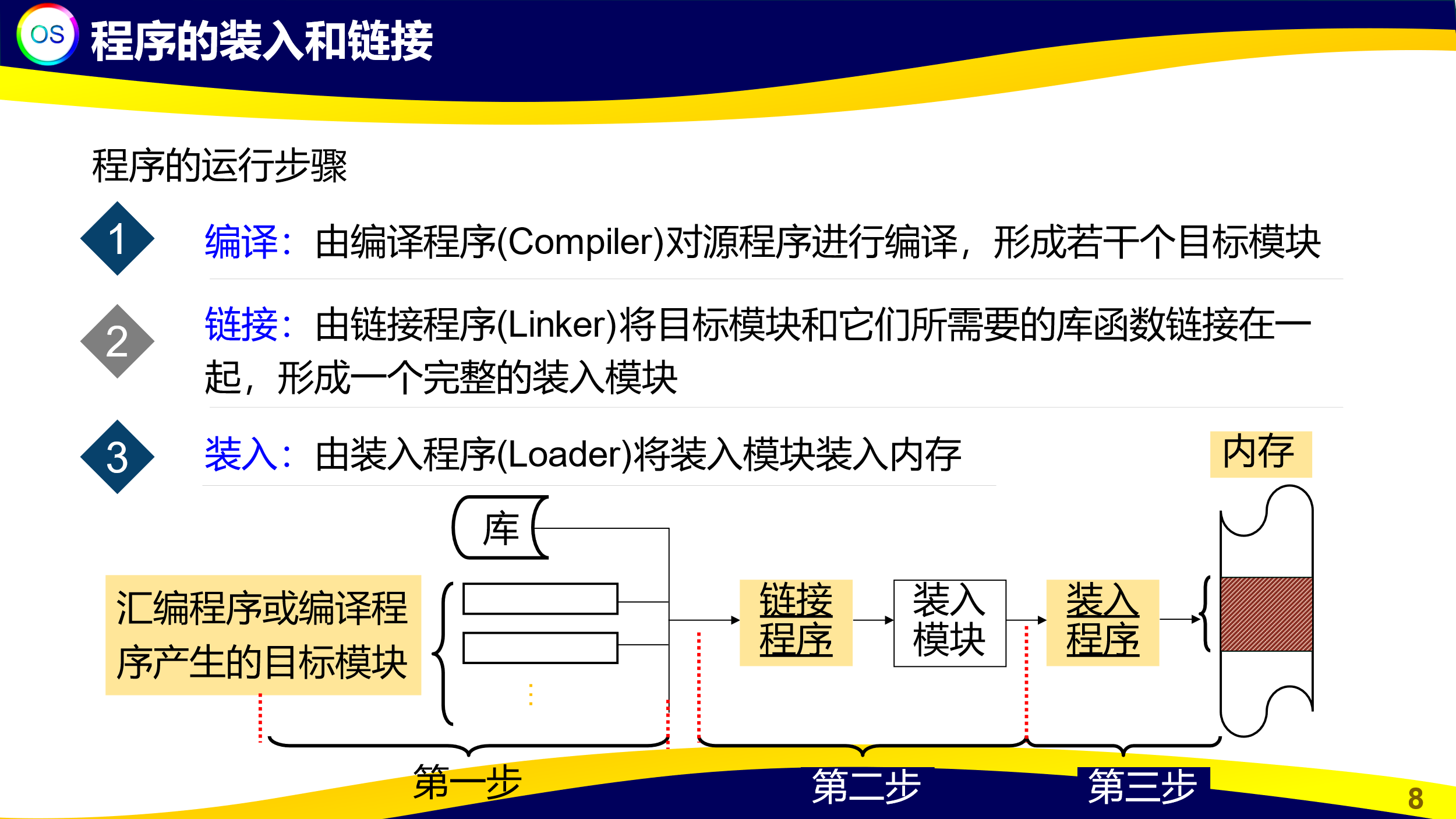
**代码解读**：

* 此代码模拟 **CPU 缓存的作用**：
  + **数组访问**：每隔 64 个元素访问一次，模拟缓存的工作方式（利用缓存行）。
  + **多次循环**：如果缓存生效，后续访问的时间将大大减少，因为数据已被预先加载到缓存。

**管层设计的关键**

1. **寄存器和缓存：加速处理**；
2. **RAM：存储当前执行的数据和指令**；
3. **磁盘缓存和外存储：高效存取大量数据**。

通过这种多层次结构设计，计算机在性能和成本之间实现平衡，确保系统在运行过程中既能快速响应又能存储大量数据。



**程序的装入和链接详细解说**

装入和链接是程序执行过程中非常重要的步骤，确保源代码通过编译、链接、装入进入内存，以便最终运行。接下来，我会按图片展示的步骤详细解释，并给出示例代码和分析。

**1. 编译（Compilation）**

* **过程**：编译器将源代码（如C代码）转换为目标代码（Object File），形成若干目标模块。
* **目标模块**：这是未完全独立的二进制文件，仍需与其他模块和库函数链接。

**示例：**

bash

复制代码

gcc -c program1.c -o program1.o

gcc -c program2.c -o program2.o

以上命令将program1.c和program2.c分别编译成program1.o和program2.o。

**2. 链接（Linking）**

* **过程**：将若干个目标模块和库文件链接成一个完整的可执行文件。
* **静态链接**：将所有代码和库打包进可执行文件，如常见的.exe。
* **动态链接**：一些库函数在程序运行时才装载，如Linux下的glibc。

**示例：**

gcc program1.o program2.o -o final\_program

将program1.o和program2.o链接成最终可执行的final\_program。

**3. 装入（Loading）**

* **过程**：由装入程序（Loader）将链接后的程序装入内存。这个步骤通常由操作系统完成，在用户运行程序时自动进行。
* **静态装入**：程序的所有部分在运行前都装载到内存中。
* **动态装入**：程序的某些部分在需要时才加载，例如共享库。

**内存示例：** 当程序被加载后，它会被映射到内存的不同区域，如：

* **代码段**：存储指令代码
* **数据段**：存储全局和静态变量
* **堆和栈**：堆用于动态分配的内存，栈用于函数调用

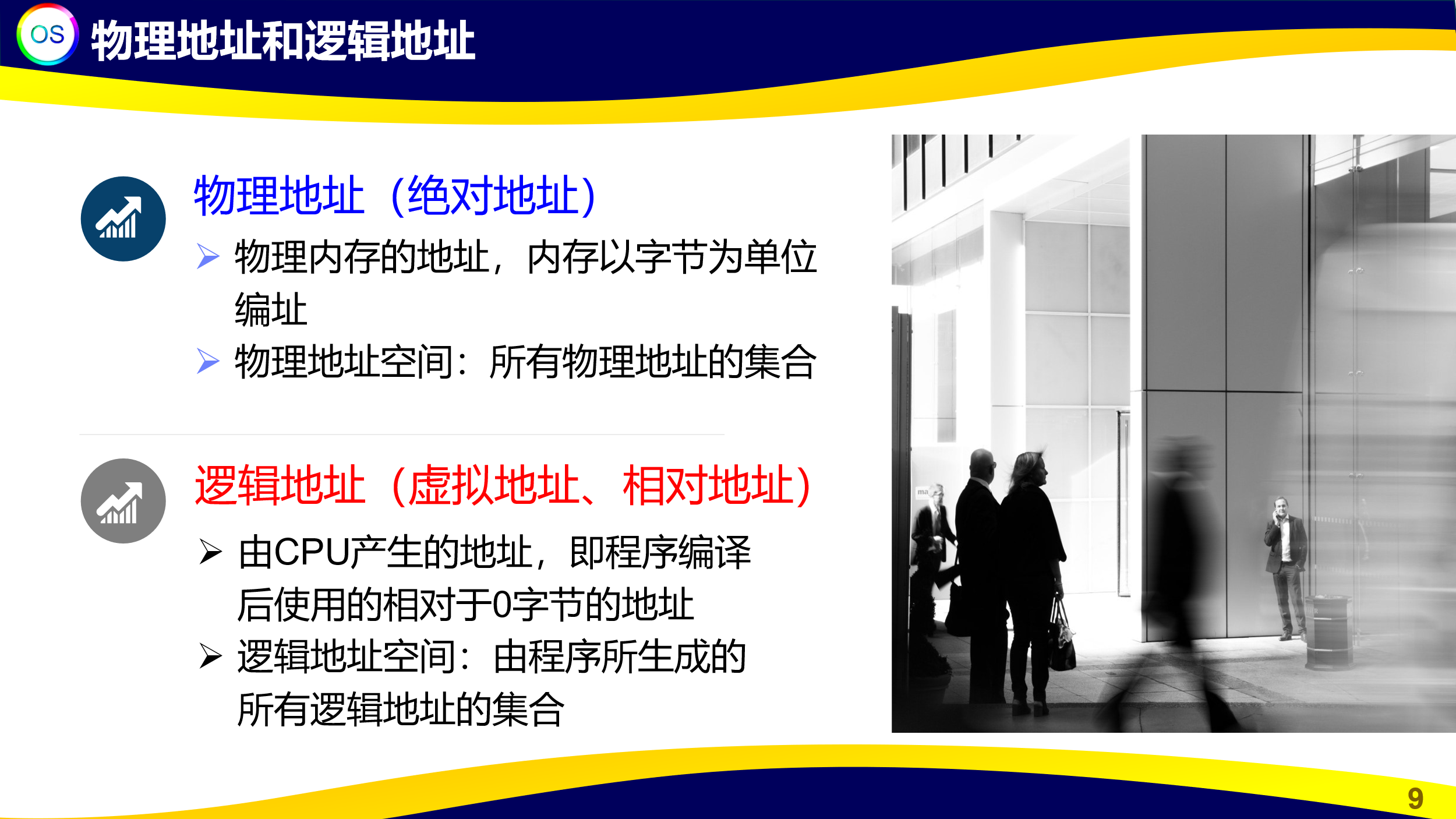
**示意图分析**

根据展示的示意图：

1. **第一步：编译**。将源代码生成一个或多个目标模块。
2. **第二步：链接**。将目标模块与库函数链接，形成一个完整的装入模块。
3. **第三步：装入**。操作系统将装入模块载入内存，以便执行。

**总结：为何这些步骤必要？**

* **提高效率**：分阶段编译和链接可以降低开发复杂度。
* **模块化设计**：通过链接，程序可以重用库函数，减少代码冗余。
* **动态链接的灵活性**：允许共享库更新而无需重编译程序。



**物理地址和逻辑地址的详细解说：**

**1. 物理地址（Physical Address）**

* **定义**：物理地址是计算机内存中实际使用的地址，是 CPU 和内存硬件之间直接交互的地址。每一条物理地址都是以字节为单位进行编排的。
* **地址空间**：所有物理地址的集合称为**物理地址空间**。这部分地址由硬件控制，无法由用户程序直接访问和操控。
* **例子**：如果内存的某个数据块存储在 0x1000 这个位置上，这个 0x1000 地址就是物理地址。
* **硬件关联**：在系统运行中，物理地址是通过内存管理单元（MMU）和硬件电路实现的。

**物理地址的特点：**

* 访问速度快，能够直接与 CPU 和内存之间通信。
* 计算机启动后，所有设备和程序都依赖于物理地址来定位存储在内存中的数据。

**2. 逻辑地址（Logical Address）**

* **定义**：逻辑地址是由 CPU 在程序运行时生成的地址，也被称为**虚拟地址**。在程序编译和执行过程中，逻辑地址被用来描述数据和指令的存储位置。
* **地址空间**：逻辑地址空间包含所有程序在运行时的虚拟地址，并且逻辑地址由操作系统映射到物理地址。

**例子**：在用户程序运行时，可能会使用逻辑地址 0x0010，但这个地址最终会映射到物理内存中的地址，比如 0x1000。

**3. 物理地址和逻辑地址之间的映射**

* 逻辑地址并不直接对应于物理地址。在现代操作系统中，CPU 生成逻辑地址后，\*\*内存管理单元（MMU）\*\*将其转换为物理地址。这种转换通常涉及页表或段表，确保地址映射的正确性和安全性。

**转换流程：**

* + CPU 生成逻辑地址。
  + MMU 检查页表，将逻辑地址转换为物理地址。
  + 若地址未被映射或无法访问，则会触发缺页中断（page fault）。

**4. 为什么需要逻辑地址？**

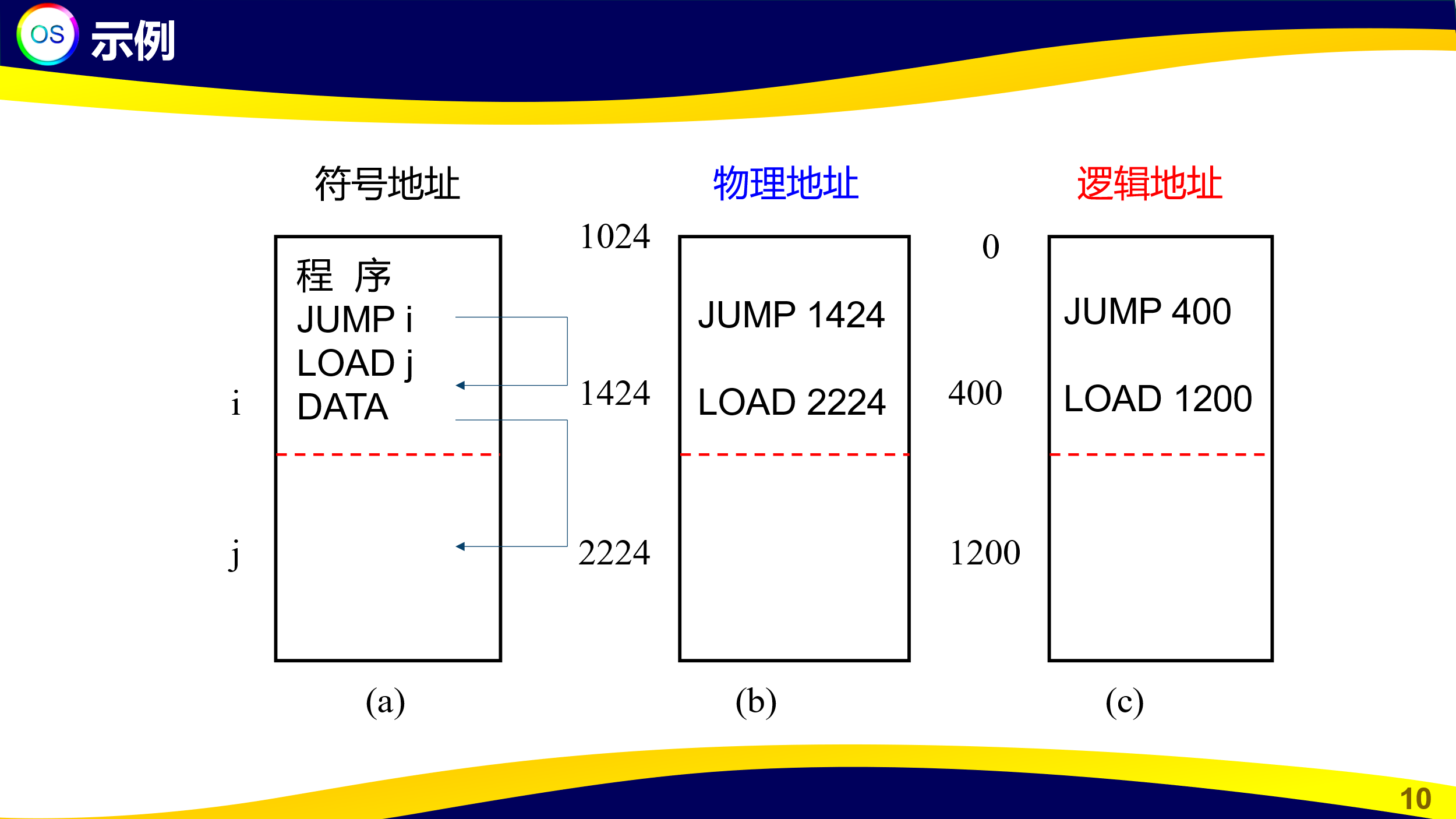
* **隔离与安全**：逻辑地址为不同进程提供独立的地址空间，避免一个进程访问其他进程的内存。
* **虚拟内存支持**：逻辑地址允许操作系统支持虚拟内存，使程序可以运行在比实际物理内存更大的地址空间中。
* **灵活性和重定位**：通过逻辑地址的映射，操作系统可以将程序和数据存放在不同的物理地址，从而提高内存利用率。

**5. 逻辑地址与物理地址的区别**

| **特性** | **逻辑地址** | **物理地址** |
| --- | --- | --- |
| **生成方式** | 由 CPU 生成 | 内存硬件生成 |
| **地址空间** | 程序运行时的地址空间 | 实际内存中的地址空间 |
| **转换** | 需要 MMU 将其映射到物理地址 | 无需转换 |
| **用途** | 用户程序使用的地址 | CPU 访问内存数据时的地址 |
| **安全性** | 提供进程隔离和虚拟内存支持 | 没有虚拟化层，直接指向内存 |

**6. 总结**

物理地址和逻辑地址的使用是现代计算机系统中内存管理的重要部分。逻辑地址为程序提供了抽象的地址空间，确保运行时的隔离与灵活性，而物理地址则负责实际数据的存储和访问。通过 MMU 的映射机制，系统可以高效、安全地进行地址转换，实现多任务操作系统的平稳运行。



**1. 符号地址（Symbolic Address）**

符号地址指的是程序在编写阶段使用的变量名或标签，如 i 和 j，它们代表存储或操作位置。这些地址在源代码中便于程序员理解，但无法直接用于内存寻址。

**示例：**

c

复制代码

int i = 10;

int j = i + 5;

这里，i 和 j 就是符号地址。

**2. 逻辑地址（Logical Address / Virtual Address）**

逻辑地址是**编译器或汇编器**在将符号地址转换后生成的地址，它是相对于某个程序开始的**相对地址**。CPU 执行时会生成这种逻辑地址。逻辑地址空间是进程在运行时的独立地址空间，由操作系统提供的虚拟内存管理实现。

**示例：** 在图中：

* JUMP i 映射成逻辑地址 JUMP 400。
* LOAD j 映射成 LOAD 1200。

这些逻辑地址是程序内部的偏移地址，它们与物理地址无关。

**3. 物理地址（Physical Address）**

物理地址指的是数据或指令在实际内存中的位置，由 CPU 通过内存管理单元（MMU）进行逻辑地址到物理地址的映射。

**转换关系：**

* **逻辑地址 400** 转换为 **物理地址 1424**。
* **逻辑地址 1200** 转换为 **物理地址 2224**。

在这个过程中，操作系统通过段表、页表等结构实现逻辑地址到物理地址的映射，确保每个进程不会干扰其他进程的内存空间。

**4. 地址转换过程**

程序运行过程中的地址转换可以分为以下几个步骤：

1. **符号地址 → 逻辑地址：**  
   编译器将符号地址转换为逻辑地址，这些地址是相对于程序开始位置的偏移量。
2. **逻辑地址 → 物理地址：**  
   操作系统通过页表或段表将逻辑地址映射到实际的物理地址。不同进程的逻辑地址可能相同，但它们会映射到不同的物理地址，确保地址空间隔离。

**5. 例子解释**

在上图的示例中：

* 在符号地址阶段，JUMP i 和 LOAD j 分别指向一些数据位置。
* 这些符号地址在编译后成为逻辑地址，例如 JUMP 400 和 LOAD 1200。
* 最后，逻辑地址被操作系统映射为物理地址，例如 JUMP 1424 和 LOAD 2224，这些地址是内存中的真实位置。

**6. 为何需要多级地址系统？**

* **灵活性：** 允许程序在内存中的任何地方运行，不需要在编写时指定具体的物理位置。
* **安全性：** 每个进程都有独立的逻辑地址空间，避免了进程之间的干扰。
* **内存管理：** 操作系统可以高效地管理内存，通过分页或分段实现内存的虚拟化。

**代码示例：简单地址映射模拟**

c

复制代码

#include <stdio.h>

#define BASE\_ADDRESS 1000 // 假设物理地址基地址

int logical\_to\_physical(int logical\_address) {

return BASE\_ADDRESS + logical\_address; // 简单的地址映射

}

int main() {

int logical\_address = 400;

int physical\_address = logical\_to\_physical(logical\_address);

printf("逻辑地址: %d -> 物理地址: %d\n", logical\_address, physical\_address);

return 0;

}

**运行结果：**

makefile

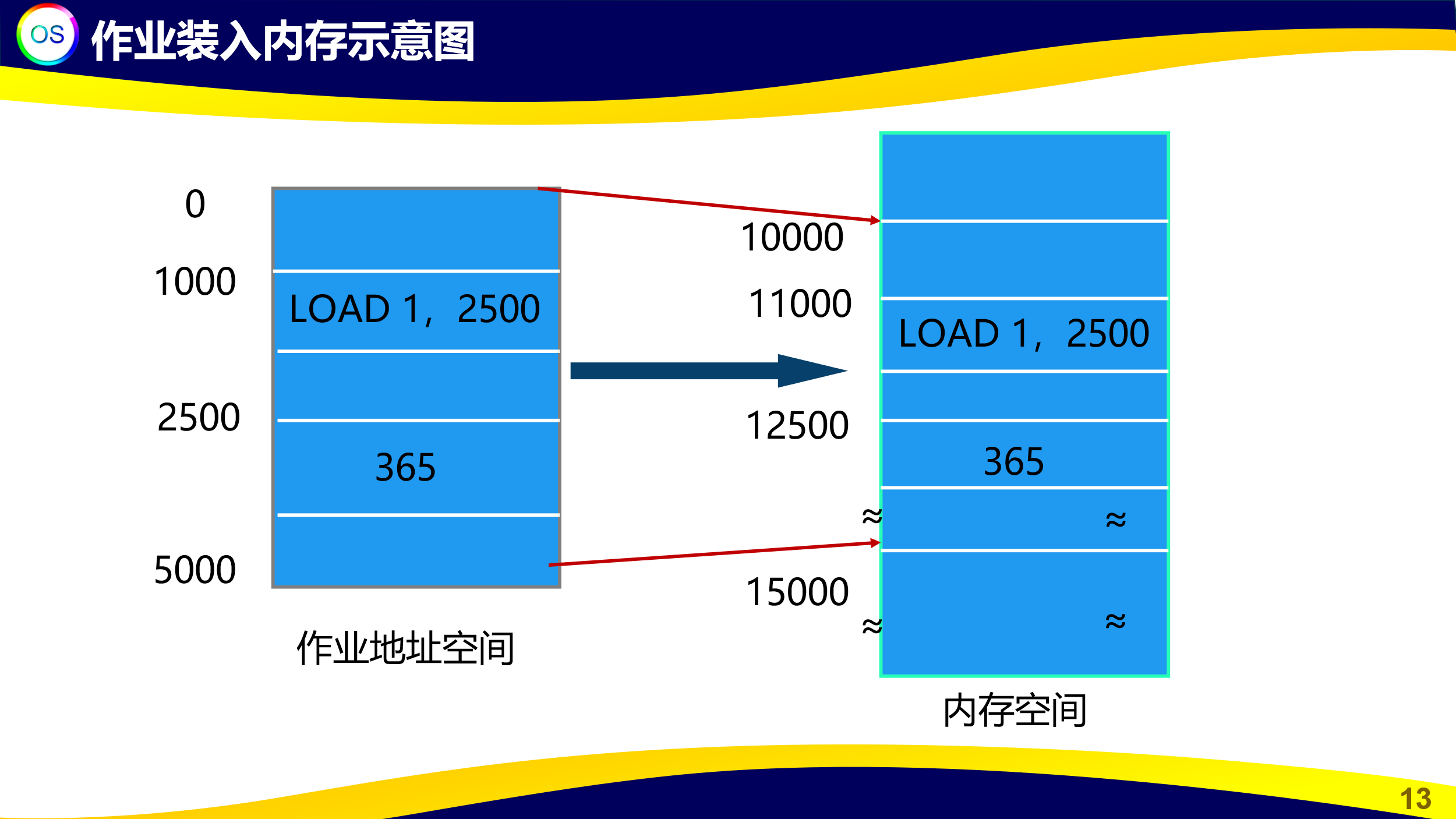
复制代码

逻辑地址: 400 -> 物理地址: 1400

这个例子展示了如何将逻辑地址转换为物理地址。在实际系统中，地址映射可能复杂得多，需要通过页表或段表实现。

**总结**

符号地址、逻辑地址和物理地址之间的关系是现代计算机系统中程序运行的核心。符号地址便于编程，逻辑地址提供进程隔离，而物理地址则对应内存中的实际位置。通过地址转换机制，系统能够在保证安全性和隔离性的同时，高效管理内存资源。



在操作系统中，装入（loading）是将程序代码和数据从磁盘装载到内存中的过程。根据装入时地址的确定方式，可以分为**绝对装入、可重定位装入和动态运行时装入**。下表将这些不同的装入方式进行了对比：

| **装入方式** | **特点** | **优缺点** |
| --- | --- | --- |
| **绝对装入方式** | - 编译时地址使用绝对地址  - 程序被修改后必须重新编译 | 优点：实现简单  缺点：不灵活，容易浪费内存 |
| **可重定位装入方式** | - 编译时使用相对地址  - 在装入时根据起始位置完成静态重定位 | 优点：灵活，减少浪费  缺点：需要一定硬件支持 |
| **动态运行时装入方式** | - 程序运行时完成地址重定位  - 也称为动态重定位 | 优点：高效利用内存，适合多任务系统  缺点：实现复杂，需要更多开销 |

**1. 绝对装入方式**

在绝对装入方式中，程序的地址在编译时就已经确定。当程序需要运行时，操作系统直接将程序加载到指定的内存地址。如果程序代码或数据发生修改，必须重新编译。

**例子**：  
假设某程序在地址 0x1000 处运行，编译时所有引用都是基于此绝对地址。如果该地址在其他任务中被占用，则无法装入。

**适用场景**：

* 嵌入式系统等固定结构的环境。

**2. 可重定位装入方式**

这种方式下，编译后的程序包含的是相对地址，在装入内存时会根据实际装入地址进行重定位。这种重定位是在程序加载时一次性完成，称为**静态重定位**。

**例子**：

* 如果程序的目标代码起始地址相对为 0，在装入时操作系统将程序放在地址 0x3000，则程序的所有引用会偏移 0x3000。

c

复制代码

// 伪代码：重定位示例

int base\_address = 0x3000;

int \*ptr = (int \*)(relative\_addr + base\_address);

**优点**：提高了内存的利用率，多个程序可以装载到不同的内存区域。

**3. 动态运行时装入方式**

动态装入方式在程序运行时完成重定位，称为**动态重定位**。地址转换由操作系统的内存管理单元（MMU）和地址转换表实现。

**例子**：

* 当程序访问逻辑地址 0x0400，操作系统根据页表将其转换为物理地址 0x3400。

c

复制代码

// 简单示例：动态转换逻辑地址到物理地址

int logical\_address = 0x0400;

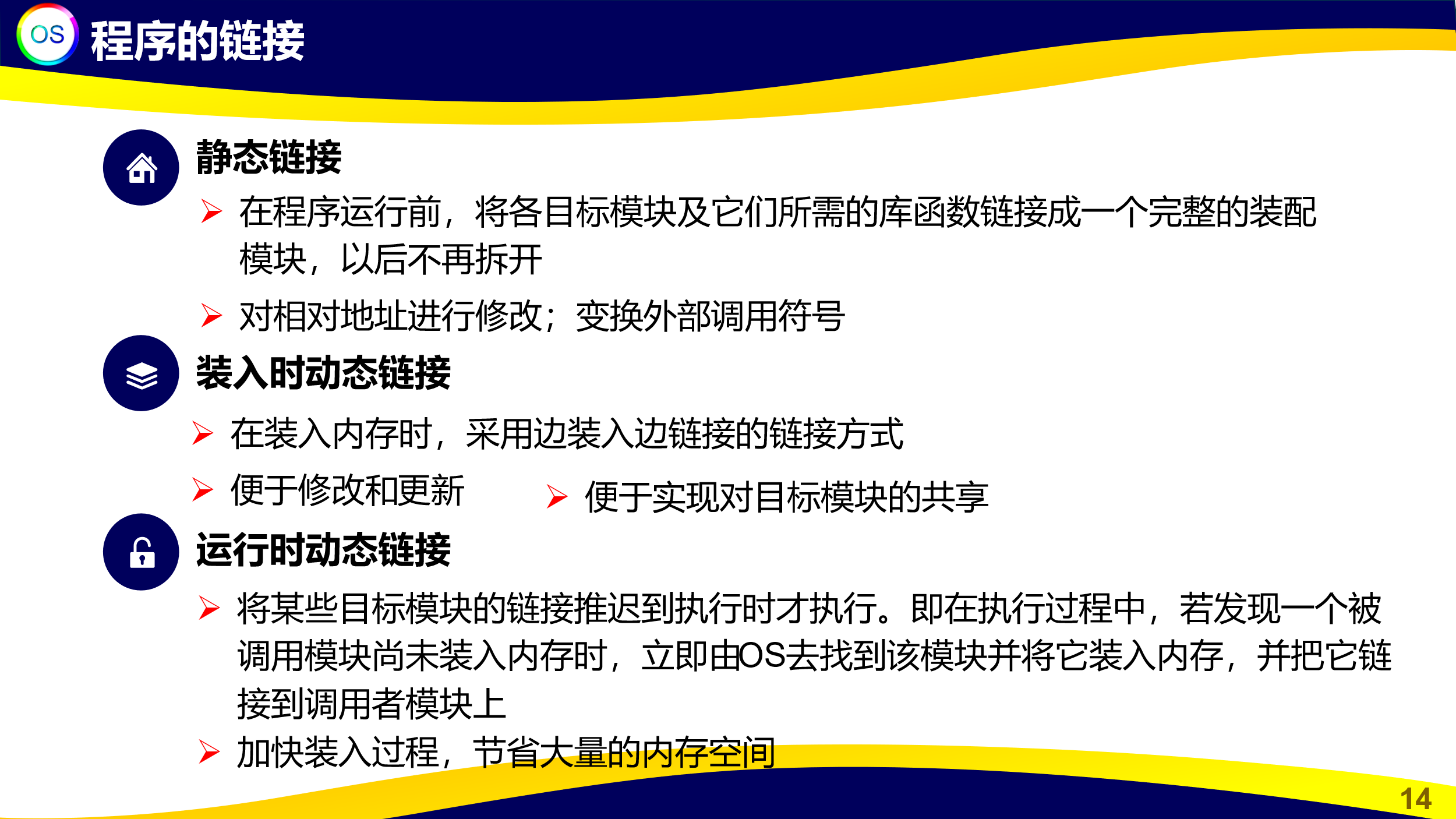
int physical\_address = page\_table[logical\_address / PAGE\_SIZE] + (logical\_address % PAGE\_SIZE);

**优点**：实现灵活的内存分配，支持动态库加载。  
**缺点**：需要硬件支持（如 MMU），并带来额外的开销。

**装入方式的意义与选择**

1. **绝对装入**适用于嵌入式系统或资源有限的环境。
2. **可重定位装入**适用于多任务系统，可以减少内存浪费。
3. **动态运行时装入**是现代操作系统常用的方式，提高了系统的灵活性和效率。

通过这些不同的装入方式，操作系统可以更好地管理内存，支持复杂的多任务处理和动态库机制，提高系统性能与资源利用率。



**程序的静态与动态链接：详细解说**

**1. 静态链接**

* **定义**：静态链接是在程序运行之前，将所有目标模块和所需的库函数预先组合成一个完整的可执行文件。这意味着，所有需要的符号都在编译链接阶段被解析和嵌入。
* **特点**：
  + 所有的符号地址在编译时确定。
  + 编译后的程序独立性强，不需要外部库的支持，适合脱机环境。
  + 文件体积较大，因为每个可执行文件包含了所有需要的库代码。

**例子**：

bash

复制代码

gcc -o program main.o utils.o -static

此命令会将 main.o 和 utils.o 静态链接，生成一个独立的可执行文件。

**2. 装入时动态链接**

* **定义**：装入时动态链接是指在程序被装载进内存的过程中，动态链接器（如 Linux 的 ld.so）将程序与共享库进行链接。
* **特点**：
  + 在程序加载时完成符号解析，这允许库的更新而无需重新编译程序。
  + 适用于系统库频繁更新的场景，减少存储空间的占用。

**例子**：

bash

复制代码

gcc -o program main.o -L/lib -lmylib

当程序运行时，系统会查找 libmylib.so 并在加载过程中完成链接。

**3. 运行时动态链接**

* **定义**：某些库模块的链接推迟到程序运行期间，即执行过程中才根据需要加载库。
* **特点**：
  + 提供更灵活的模块加载，适合插件系统。
  + 在执行过程中根据需要加载模块，减少了初始内存占用。
  + 使用操作系统的 dlopen 和 dlsym 实现。

**例子**：

c

复制代码

#include <dlfcn.h>

int main() {

void \*handle = dlopen("libmylib.so", RTLD\_LAZY);

if (!handle) {

fprintf(stderr, "%s\n", dlerror());

return 1;

}

// 使用符号解析的函数

dlclose(handle);

return 0;

}

上述代码展示了如何在运行时动态加载一个库，并解析其中的符号。

**静态与动态链接的对比**

| **特点** | **静态链接** | **动态链接** |
| --- | --- | --- |
| **加载时间** | 较快，因为已经完成了符号解析 | 较慢，需要解析符号和链接库 |
| **文件大小** | 较大，包含所有所需库的代码 | 较小，依赖共享库 |
| **更新成本** | 高，需要重新编译 | 低，只需更新共享库 |
| **内存占用** | 每个程序都有一份库代码 | 多个程序共享库 |

**为何需要动态链接？**

1. **节省存储空间**：共享库减少了重复代码的存储。
2. **易于维护与更新**：无需重新编译程序，只需更新库。
3. **灵活性**：支持在运行时加载不同版本的模块。

**场景举例：**

* **插件系统**：如浏览器插件，只有在用户启用某个插件时才加载相应模块。
* **操作系统更新**：Linux 系统更新时，只需更新共享库，而无需更新所有使用该库的应用。

**总结**

动态链接提供了灵活性和节省资源的优势，特别是在现代系统中，广泛使用动态链接来提高维护性和降低存储成本。然而，静态链接在某些脱机或独立环境中仍然有其价值，特别是需要高性能和独立性的场景。

让我们用一个现实生活中的比喻来理解“静态链接”和“动态链接”的区别和应用。

**1. 静态链接：预先准备好的一次性大餐**

* **比喻**：就像你参加一个野餐活动时，为了不再跑去商店买食物，你提前准备好了所有的菜品、零食，并把它们都打包在一起。
* **特点**：
  + **提前打包**：所有的代码和库函数（就像你的食物）都已经提前准备好了。
  + **运行时独立**：你在野餐时不需要再额外准备任何东西（程序不依赖外部资源）。
  + **缺点**：但是你携带的东西可能很重（程序体积较大），而且如果某道菜（库）有问题，你必须重新准备整个野餐篮（重新编译整个程序）。

**现实例子**：

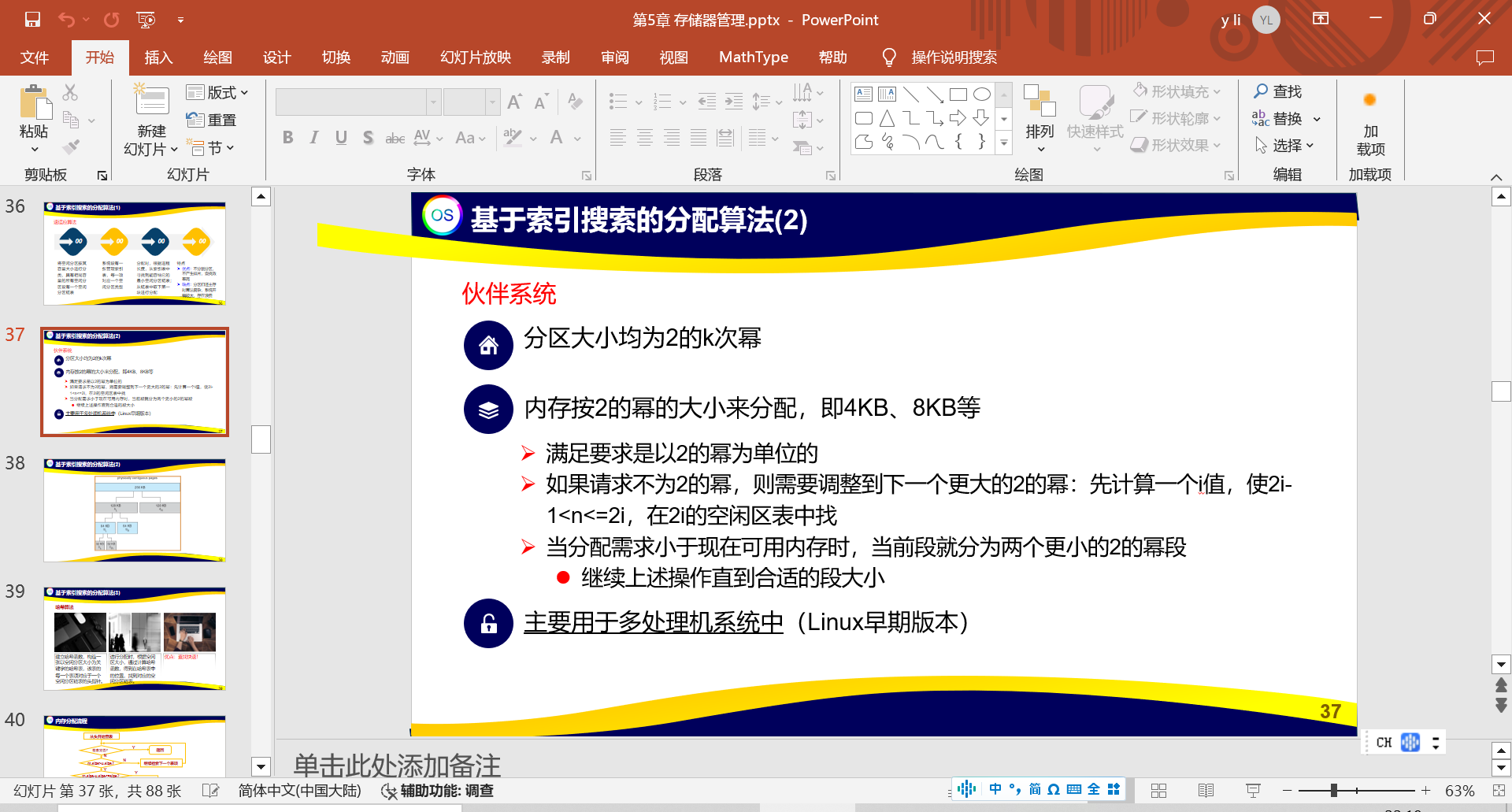
* 一些单机游戏或嵌入式设备使用静态链接的方式，这样即便离线也能正常运行。

**2. 动态链接：即时订购的外卖**

* **比喻**：动态链接就像你去参加野餐时，只带了基本的东西。需要什么特殊食物时，你随时可以用手机点外卖。
* **特点**：
  + **灵活性**：当程序需要一个特定的库时（比如某个共享函数），它会在运行时去操作系统的“共享库”中查找并使用（类似下单外卖）。
  + **节省空间**：因为多个程序可以共享同一个库，不需要每个程序都打包自己的库。
  + **缺点**：如果外卖店没开门（共享库版本不兼容），程序就可能无法正常运行。

**现实例子**：

* 现代操作系统（如 Linux 和 Windows）中的大多数软件使用动态链接，例如程序启动时会去加载系统中的 libc.so 等常用库。



**伙伴系统简介：**

* 伙伴系统是一种内存管理算法，用于动态内存分配，旨在在分配和回收内存时减少碎片化问题。
* 内存块按2的幂次分配，例如：4KB、8KB、16KB 等。
* 当程序需要的内存大小不为2的幂时，会向上取最近的2的幂大小进行分配。

**算法工作原理：**

1. **内存初始化：**
   * 内存被分为若干大小相同的区域，每个区域的大小是2的幂次方，例如8KB、16KB等。
   * 所有未分配的内存块会维护在一个空闲链表中，并按大小分类。
2. **内存分配：**
   * 请求内存时，算法寻找能够满足需求的最小2的幂次内存块。
   * 如果没有合适大小的块，会将更大的块分成两个相等的伙伴块，直到得到合适大小的块。
3. **内存释放：**
   * 当释放一个内存块时，算法检查它的“伙伴”是否也空闲。
   * 如果伙伴也是空闲的，则合并这两个伙伴块形成一个更大的块，重复此过程直到无法再合并。

**代码示例：**

c

复制代码

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define MAX\_ORDER 5 // 假设最大支持32KB的块

typedef struct Block {

int order;

struct Block\* next;

} Block;

Block\* free\_list[MAX\_ORDER + 1]; // 每一阶的空闲块链表

// 初始化内存

void init\_memory() {

for (int i = 0; i <= MAX\_ORDER; i++) {

free\_list[i] = NULL;

}

}

// 分配块

void\* allocate(int size) {

int order = ceil(log2(size)); // 找到最近的2的幂次

if (order > MAX\_ORDER) {

printf("内存请求太大！\n");

return NULL;

}

for (int i = order; i <= MAX\_ORDER; i++) {

if (free\_list[i] != NULL) { // 找到可用块

Block\* block = free\_list[i];

free\_list[i] = block->next;

// 将块分割成更小的伙伴块

while (i > order) {

Block\* buddy = (Block\*)((char\*)block + (1 << (i - 1)));

buddy->order = i - 1;

buddy->next = free\_list[i - 1];

free\_list[i - 1] = buddy;

i--;

}

return (void\*)block;

}

}

printf("没有足够的内存！\n");

return NULL;

}

// 释放块

void deallocate(void\* ptr, int order) {

Block\* block = (Block\*)ptr;

block->order = order;

// 查找伙伴并合并

while (order < MAX\_ORDER) {

char\* buddy\_addr = (char\*)block ^ (1 << order); // 找到伙伴地址

Block\* buddy = (Block\*)buddy\_addr;

// 如果伙伴在空闲链表中，则合并

Block\*\* current = &free\_list[order];

while (\*current != NULL && \*current != buddy) {

current = &(\*current)->next;

}

if (\*current == buddy) {

\*current = buddy->next; // 从空闲链表中移除伙伴

block = (block < buddy) ? block : buddy; // 更新块指针

order++;

} else {

break;

}

}

// 将合并后的块加入空闲链表

block->next = free\_list[order];

free\_list[order] = block;

}

**代码解读：**

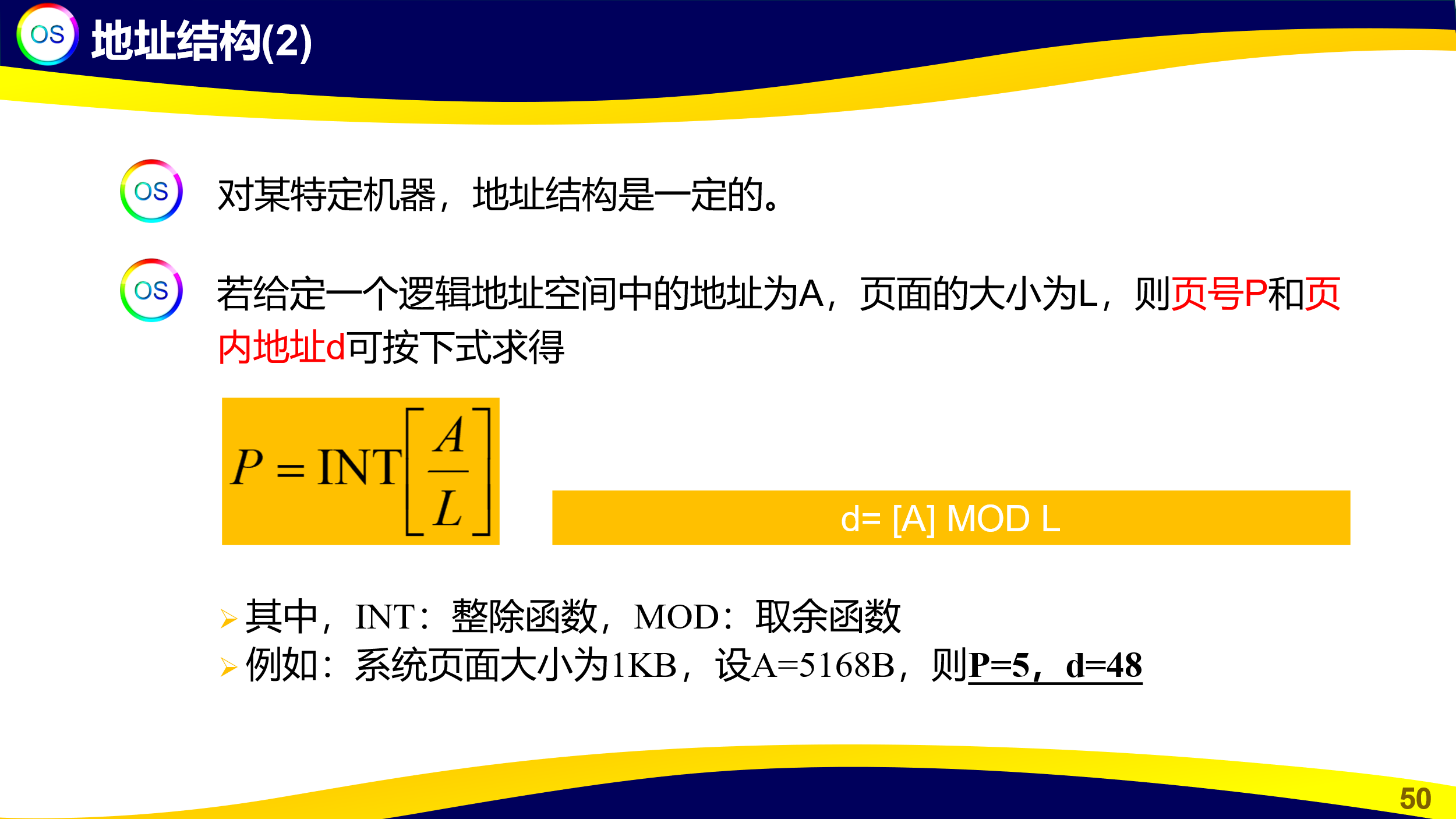
1. **初始化：** init\_memory 函数将所有的空闲链表初始化为空。
2. **内存分配：**
   * allocate 函数根据请求大小寻找最合适的内存块。
   * 如果没有直接匹配的块，会将更大的块按伙伴关系分割成更小的块，直到找到合适大小。
3. **内存释放：**
   * deallocate 函数释放内存块时，会检查伙伴是否空闲。
   * 如果伙伴空闲，则合并这两个块，直到无法再合并为止。

**伙伴系统的优势：**

* **减少碎片化：** 由于内存总是按2的幂次分配，伙伴系统能够有效减少内存碎片。
* **快速合并：** 通过简单的地址计算，伙伴系统可以快速找到并合并空闲块。
* **适用于多道处理：** 适用于需要频繁分配和释放内存的系统。

**伙伴系统的应用：**

* 早期的 **Linux** 内存管理系统。
* 实时系统或嵌入式系统中，为了减少碎片并加速内存分配。



页面大小为1KB，地址A=5168B，我们可以使用分页逻辑来计算与该地址相关的页号（P）和偏移量（d）。这是分页存储管理中的常见计算。

**计算步骤：**

* **已知信息：**
  + 页面大小 =1= 1=1 KB =1024= 1024=1024 B
  + 逻辑地址 A=5168A = 5168A=5168 B

**计算页号（P）：**

页号 PPP 表示地址属于第几页，可以通过将逻辑地址除以页面大小来获得：

P=A页面大小=51681024=5P = \frac{A}{页面大小} = \frac{5168}{1024} = 5P=页面大小A​=10245168​=5

所以，页号 P=5P = 5P=5。

**计算页内偏移（d）：**

偏移量 ddd 是地址在该页中的位置，可以通过取地址与页面大小的余数来计算：

d=A mod 页面大小=5168 mod 1024=48d = A \bmod 页面大小 = 5168 \bmod 1024 = 48d=Amod页面大小=5168mod1024=48

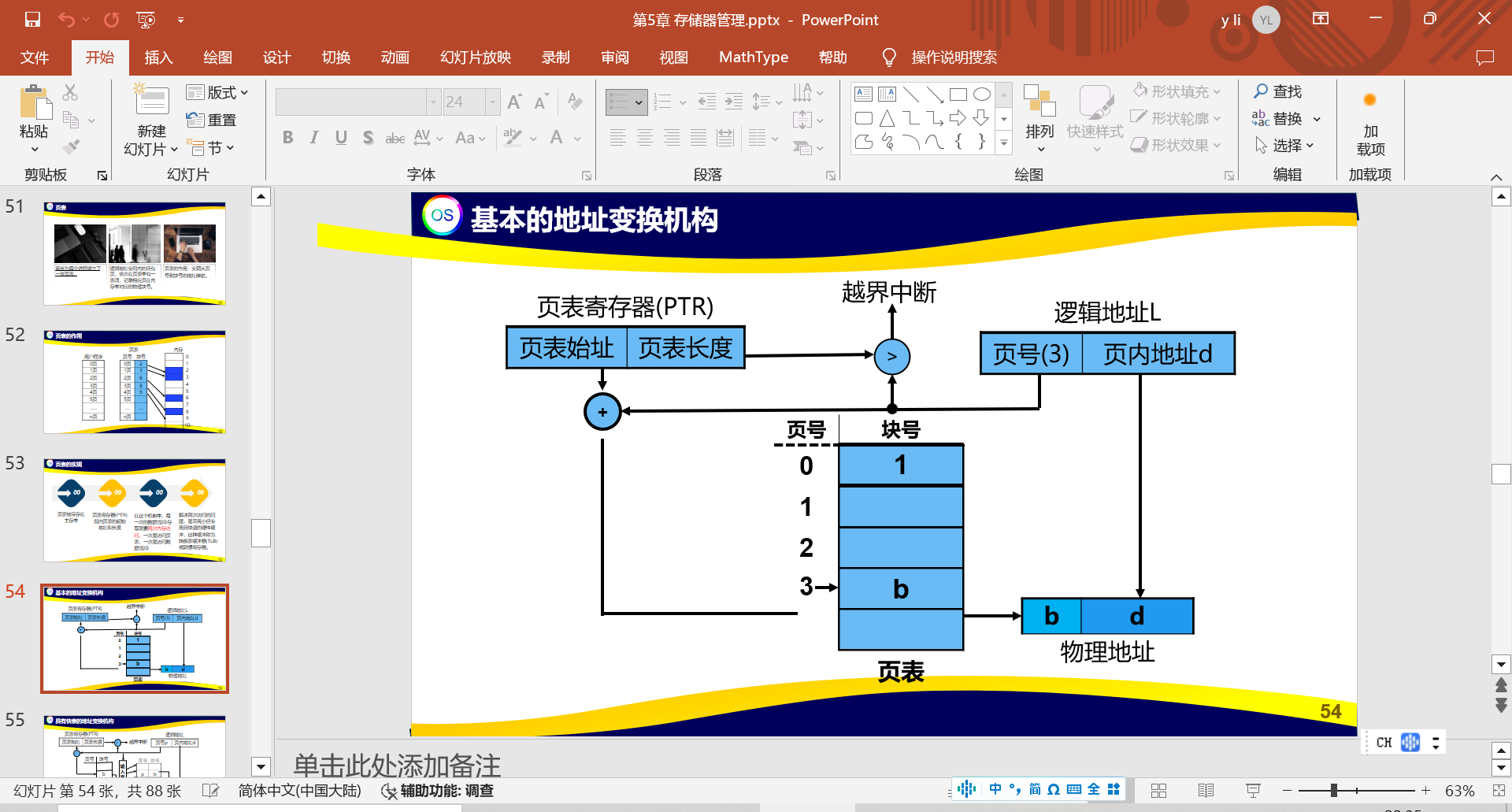
因此，偏移量 d=48d = 48d=48。

**解释：**

1. 逻辑地址 5168B 表示系统中的一个虚拟地址，在分页系统中必须将其转换为页号和页内偏移量，以便正确访问内存。
2. **页号 P=5P=5P=5** 说明这个逻辑地址对应于**第5页**。
3. **偏移量 d=48d=48d=48** 表示在该页的第 48 字节处存储了具体的数据。

**总结：**

在分页系统中，将逻辑地址 5168B 分解为页号和偏移量的过程使得虚拟地址可以映射到物理内存页，从而实现内存的高效管理。这种方式可以减少内存碎片，同时支持多任务系统中不同进程的隔离与保护。



**1. 背景：逻辑地址与物理地址**

* **逻辑地址**（虚拟地址）：这是程序生成的地址，表示程序中的一个变量或指令的位置。逻辑地址在程序的视角中是连续的。
* **物理地址**：这是内存中的真实地址，表示数据存储在内存中的实际位置。

因为程序员只关心逻辑地址，而操作系统负责将这些逻辑地址映射到物理地址。

**2. 地址变换的主要过程**

1. **页表寄存器（PTR）**：存储页表的起始地址和长度信息。它帮助操作系统找到页表在内存中的具体位置。
2. **逻辑地址的组成**：
   * **页号**（Page Number）：表示该逻辑地址所属的页。
   * **页内偏移量（d）**：表示该逻辑页内的数据具体位置。
3. **通过页表查找物理块**：
   * **页号 P** 用来访问页表，找到该页号对应的物理块号 bbb。
   * 物理块号 bbb 加上偏移量 ddd，即得到了完整的物理地址。

**3. 示例说明**

假设：

* 系统页面大小为 **1KB**（1024字节）。
* 页表的每一页映射到一个内存块。
* 假如逻辑地址 **L** 为 (3, 48)，则：
  + **页号 = 3**，表示访问页表中的第3页。
  + **偏移量 d = 48**，表示该页内的第48个字节。

1. **转换步骤**：
   * 页表查询：根据页号3找到对应的物理块号（如1号块）。
   * 偏移量48加到物理块的起始地址上，形成最终的物理地址。

**4. 为何采用分页？**

分页技术让程序可以不必连续分配内存，提高内存利用率，并且简化了地址管理。不同于分段，分页是固定大小的内存块，有助于减少内存碎片。

**5. 代码实现分页转换**

c

复制代码

#include <stdio.h>

#define PAGE\_SIZE 1024 // 定义页面大小为1KB

// 模拟页表

int page\_table[4] = {1, 2, 3, 4}; // 每个页号映射到对应的物理块号

// 地址转换函数

int address\_translation(int page\_number, int offset) {

if (page\_number >= 4) {

printf("Invalid page number!\n");

return -1;

}

int physical\_block = page\_table[page\_number];

int physical\_address = physical\_block \* PAGE\_SIZE + offset;

return physical\_address;

}

int main() {

int page = 3; // 页号

int offset = 48; // 偏移量

int physical\_address = address\_translation(page, offset);

if (physical\_address != -1) {

printf("Physical Address: %d\n", physical\_address);

}

return 0;

}

**6. 代码解读**

* **页表数组**：page\_table[4] 模拟页表，每个逻辑页号映射到一个物理块号。
* **address\_translation()**：该函数将逻辑地址拆解为页号和偏移量，通过页表查找物理块号，计算出最终物理地址。
* **主函数**：指定一个逻辑地址（页号3，偏移量48），调用地址转换函数，打印出对应的物理地址。

**7. 总结**

通过分页，操作系统能够高效管理内存。即使程序的逻辑地址是连续的，实际存储在内存中的物理地址却可以是非连续的。这种映射让系统能够更加灵活地分配内存，并减少内存碎片。



有效访问时间 (EAT, Effective Access Time)\*\*的计算方法，这是一个用于评估存储系统中访问时间效率的重要公式。

**1. EAT 公式的基本含义**

在存储系统中，当使用**页表和快表**（TLB, Translation Lookaside Buffer）时，我们希望通过快表的高命中率来减少访问内存的总时间。公式如下：

EAT=λ⋅a+(t+λ)⋅(1−a)+tEAT = \lambda \cdot a + (t + \lambda) \cdot (1 - a) + tEAT=λ⋅a+(t+λ)⋅(1−a)+t

其中：

* λ\lambdaλ：查找快表所需的时间
* ttt：访问内存所需的时间
* aaa：快表的命中率（即找到页号的概率）

**2. 公式推导**

* **命中情况**：若快表命中，即在快表中找到对应的页表项，只需花费查找快表的时间 λ\lambdaλ 加上访问内存的时间 ttt。  
  所以，这部分时间贡献为 λ+t\lambda + tλ+t。
* **未命中情况**：若快表未命中，则需要查找页表，耗费的时间为 t+λt + \lambdat+λ，加上访问内存的时间 ttt，总时间为 t+λ+t=2t+λt + \lambda + t = 2t + \lambdat+λ+t=2t+λ。
* **总的 EAT**：根据命中率 aaa，可以得到：

EAT=a⋅(λ+t)+(1−a)⋅(2t+λ)EAT = a \cdot (\lambda + t) + (1 - a) \cdot (2t + \lambda)EAT=a⋅(λ+t)+(1−a)⋅(2t+λ)

**3. EAT 公式的意义**

该公式反映了在**不同的命中率和访问时间组合**下，系统的总体访问效率。提高**快表的命中率 aaa** 或减少 **查找快表的时间 λ\lambdaλ** 是优化系统性能的重要手段。

**4. 局部性与性能优化**

在现实系统中，存储系统通过提高\*\*局部性（locality）\*\*来增加快表的命中率。幻灯片中提到快表的局部性约为90%，这意味着大部分内存访问都能够在快表中找到，从而显著缩短访问时间。

**5. 总结**

EAT 公式是计算机系统中分析存储器性能的关键工具之一。提高快表命中率，优化内存访问路径，减少不必要的页表查找时间，是系统设计中的重要环节。

**第59页 - "两级页表结构"概述**

**内容说明**：

* 这张幻灯片介绍了两级页表的概念，展示了如何通过两级分页系统将逻辑地址映射为物理地址。
* 页面左侧显示的是“外层页表”（outer page table），每个入口指向内层页表（page of page table）。
* 页表索引和偏移量用于确定物理地址的位置。

**详细解说**：

* **两级分页的目的**：分页表用于存储进程的页号和物理地址的映射关系。单层分页表在进程地址空间较大时会消耗大量内存，因此多级分页将分页表分成更小的部分，减少了不必要的内存占用。
* **映射过程**：
  1. 将逻辑地址分为三个部分：p1、p2和偏移量d。
  2. p1用于索引外层页表，找到对应的内层页表。
  3. p2用于在内层页表中查找实际的物理页框。
  4. 偏移量d则定位到该页框中的具体字节。
* **案例分析**：假设逻辑地址为p1=1，p2=2，d=50，首先通过p1找到对应的内层页表，再通过p2确定物理页框，最后加上偏移量得到最终物理地址。

**第60页 - "两级页表结构的逻辑图"**

**内容说明**：

* 图示展示了逻辑地址的分解过程及其在两级页表结构中的映射。
* 使用逻辑地址的三个部分（p1、p2和d）逐步索引页表，最终获得物理地址。

**详细解说**：

* **逻辑地址分解**：逻辑地址分为三级部分，依次代表外层页表索引（p1）、内层页表索引（p2）和页内偏移量（d）。
* **映射过程的解构**：
  + p1决定了在哪个内层页表中进行查找。
  + p2在内层页表中选择物理页框。
  + 偏移量d直接加到物理页框的基地址上。
* **举例说明**：例如，如果p1=3，p2=5，偏移量d=100，先查找外层页表索引p1=3的位置，得到内层页表的地址，然后在内层页表中找到p2=5的页框位置，最后加上偏移量得到完整的物理地址。

**第62页 - "反置页表"概念**

**内容说明**：

* 反置页表用于减少内存中的页表占用，适合大型内存系统。
* 每个物理块对应一个条目，保存当前映射到该块的页号等信息。

**详细解说**：

* **反置页表的原理**：反置页表与传统页表不同，每个物理内存块对应一个表项，用于存储该块中存放的逻辑页的信息（包括页号和所属进程ID等）。
* **优势与劣势**：
  + 优势：减少页表占用的内存空间，因为页表大小与物理内存大小相关，而非进程的逻辑地址空间。
  + 劣势：查询较慢，因为访问特定页时需要遍历反置页表来查找映射。
* **应用举例**：适用于大型服务器和虚拟机系统中，通过哈希查找方式来加速页表查询过程，以减少遍历时间。

**第63页 - "反置页表地址转换机制"**

**内容说明**：

* 该图描述了反置页表中的地址转换过程。
* 逻辑地址通过进程ID（pid）、页号（p）、偏移量（d）来转换为物理地址。

**详细解说**：

* **步骤分解**：
  1. CPU生成逻辑地址（pid、p、d），通过pid识别所属进程。
  2. 使用页号p查找反置页表，找到与该逻辑页匹配的物理页框。
  3. 偏移量d加到物理页框的基地址上，得到物理地址。
* **反置页表的应用**：适用于多进程系统，帮助操作系统节省页表空间，特别是在虚拟化环境中常见。

**1. 背景知识**

在普通页表结构中，每个进程会有一个独立的页表来记录该进程的逻辑页与物理帧的映射关系。然而，这种结构在大内存环境中会占用大量空间，因为每个进程都需要为其地址空间维护一个完整的页表。

**反置页表的出现**解决了这个问题，通过将页表条目数量与物理帧数量挂钩（而不是进程的逻辑地址空间大小）。这样，反置页表在每个物理内存帧上记录了当前存放的逻辑页，从而减少了页表占用的空间。

**2. 例子解释**

假设我们有一个简单的系统，它的**内存总共可以容纳4个物理帧**（编号为0、1、2、3），并且有\*\*两个进程（进程A和进程B）\*\*在运行。

* 每个进程各有自己的逻辑地址空间。
* 每个逻辑页的大小是1KB。

**假设**：

* 进程A的逻辑页1和逻辑页2被映射到物理帧0和1。
* 进程B的逻辑页3被映射到物理帧2。

在普通页表中，我们会为进程A和进程B分别建立两个页表，每个页表记录各自的逻辑页到物理帧的映射关系。但是在反置页表中，我们不再为每个进程单独建立页表，而是建立一个**全局的反置页表**。

**3. 反置页表的工作原理**

反置页表的每一行对应一个物理帧，记录了当前存放在该物理帧中的页的信息。每条记录包括：

* **所属进程的ID（pid）**
* **逻辑页号（page number）**
* **其他信息**

| **物理帧号** | **进程ID** | **逻辑页号** |
| --- | --- | --- |
| 0 | A | 1 |
| 1 | A | 2 |
| 2 | B | 3 |
| 3 | - | - |

在这个反置页表中：

* 物理帧0存储的是**进程A的逻辑页1**。
* 物理帧1存储的是**进程A的逻辑页2**。
* 物理帧2存储的是**进程B的逻辑页3**。
* 物理帧3当前没有被任何页使用，因此为空。

**4. 如何查找物理地址**

假设进程A想要访问逻辑页1的某个数据，那么操作系统会按照以下步骤进行转换：

1. **生成逻辑地址**：进程A请求的逻辑地址包括进程ID（A）、逻辑页号（1）和偏移量（例如100字节）。
2. **在反置页表中查找**：操作系统会遍历反置页表，找到符合条件的条目，即进程ID为A并且逻辑页号为1的条目。
3. **计算物理地址**：假设找到的条目在物理帧0中，那么物理地址就是物理帧0的起始地址加上偏移量100字节。

**查询结果**：

* 最终物理地址 = 物理帧0的起始地址 + 偏移量100

**5. 举个更通俗的例子**

可以把反置页表比作一个**图书馆的书架**，图书馆有若干个书架（物理帧），每个书架上只能放一本书（即一个逻辑页），每本书都有标签（逻辑页号和所属进程ID）。

**假设**：

* 小王（进程A）有一本编号为1的书放在书架0上。
* 小李（进程B）有一本编号为3的书放在书架2上。

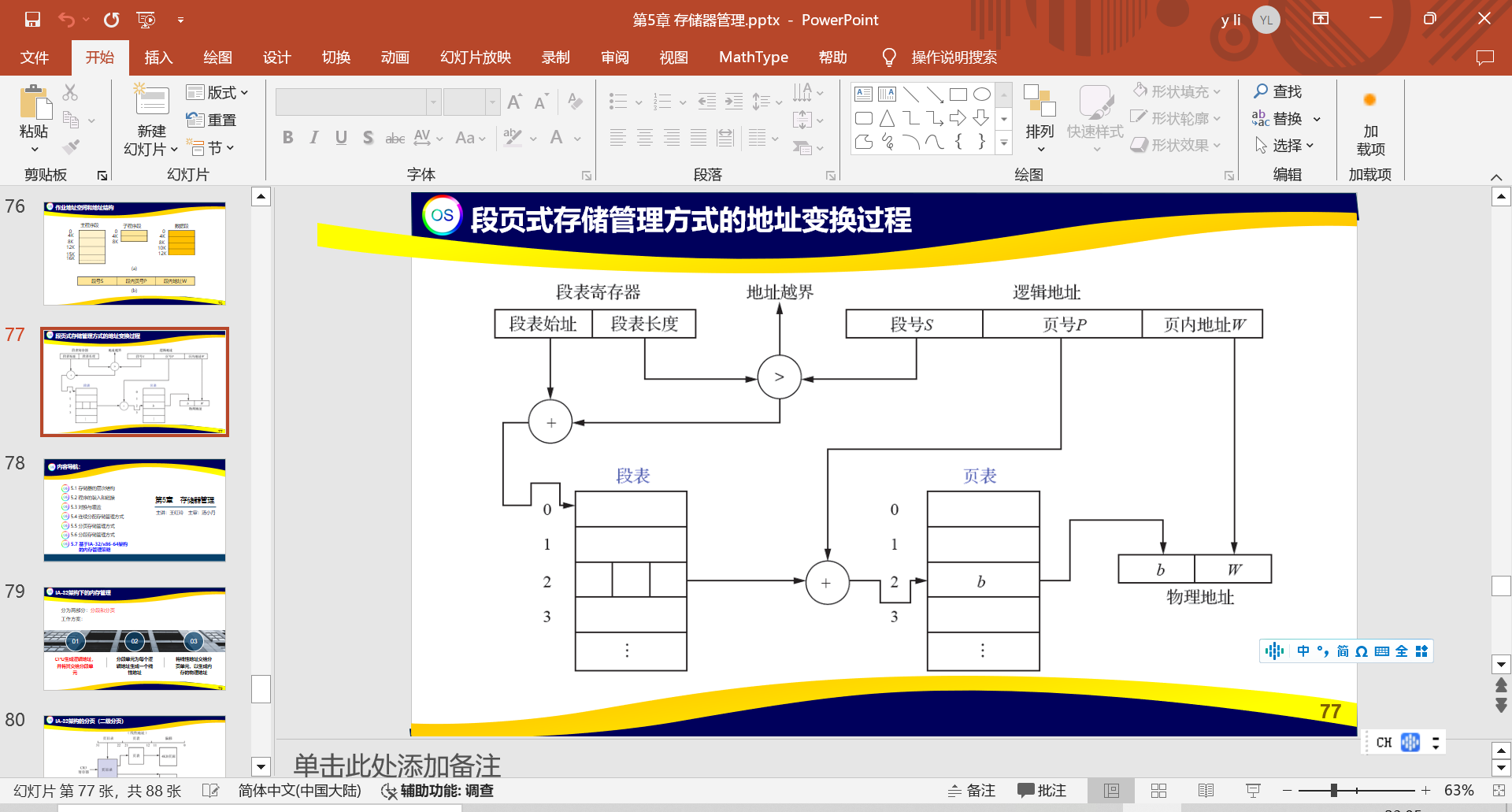
当小王来找书时，他并不是去查自己专属的目录，而是去总目录（反置页表）里查看哪个书架上放着他这本编号为1的书。找到后，他就可以拿到书（访问对应的物理地址）了。

**6. 反置页表的优势与不足**

* **优势**：
  + 节省内存：由于不需要为每个进程单独创建页表，所以节省了大量内存。
  + 适合大规模系统：在大内存环境和多进程系统下尤为有效。
* **不足**：
  + 查找速度较慢：在反置页表中查找时，需要遍历或借助哈希查找来找到目标页，查询速度相比普通页表要慢一些。
  + 实现复杂：因为每次访问都要根据进程ID和页号来查询，所以需要复杂的算法来加速查询。

**总结**

反置页表通过将页表条目与物理内存帧挂钩，解决了大型系统中页表占用过多内存的问题。虽然查找过程稍复杂，但在内存资源紧张的场景下，反置页表能显著提高内存利用效率。



在段页式管理中，逻辑地址被分成了**段号**、**页号**和**页内地址**三部分。这种结构结合了段式和页式存储管理的优点，允许更灵活的内存管理。我们来详细解读其中每个步骤的地址转换过程。

**1. 逻辑地址的组成**

逻辑地址在段页式存储中被划分为三个部分：

* **段号 SSS**：标识进程的逻辑段。
* **页号 PPP**：标识段中的页。
* **页内地址 WWW**：标识页内的偏移量。

**2. 地址转换步骤**

地址转换过程可以分为以下几个主要步骤：

1. **段表查找**：
   * 段表寄存器（Segment Table Register, STR）中存储了段表的起始地址和段表长度。
   * 系统首先检查逻辑地址中的段号 SSS 是否越界。如果段号 SSS 大于段表长度，说明访问越界，触发越界异常。
   * 如果段号 SSS 合法，则使用段号在段表中查找到对应段的起始地址和段的长度信息。
2. **段页查找**：
   * 通过段表找到段的起始地址后，加上逻辑地址中的页号 PPP，即可定位到段中相应的页表。
   * 页表中每一项对应一个物理页框的地址，通过页表查找到段内指定页的物理页框号 bbb。
3. **计算物理地址**：
   * 得到物理页框号 bbb 后，将其与页内地址 WWW 组合，形成完整的物理地址。
   * 最终的物理地址是由物理页框 bbb 的起始地址加上页内偏移 WWW 组成的。

**3. 举例说明**

假设有以下条件：

* 段表寄存器指向段表的起始位置。
* 逻辑地址为 S=2S=2S=2、P=3P=3P=3、W=100W=100W=100。

**转换过程：**

1. **查找段表**：根据段号 S=2S=2S=2，在段表中找到段 2 的页表起始地址，假设为 1000。
2. **查找页表**：根据页号 P=3P=3P=3，在段 2 的页表中找到对应的物理页框号，假设为 2000。
3. **计算物理地址**：物理页框号 2000 加上偏移量 W=100W=100W=100，得到物理地址 2100。

**总结**

这种结构的优点是：

* **灵活性**：每个段可以有不同的长度，满足不同的数据结构需求。
* **保护性**：段表提供了越界保护，防止访问不属于该段的内存。
* **共享性**：段页式结构可以方便地实现段的共享。

这种方式在大型系统中广泛应用，因为它既能满足灵活的内存分配需求，又能提供有效的内存保护机制。

**段页式存储管理方式的地址转换过程**

**内容解读：** 这页展示了段页式存储管理方式中地址转换的过程，包括段寄存器、页表和物理地址生成的步骤。段页式存储将内存分为段和页两级管理。

1. **段号 (S)**：在逻辑地址中标识要访问的段。
2. **段表**：包含每个段的起始地址和长度，用于查找该段在内存中的位置。
3. **页号 (P)** 和 **页内地址 (W)**：进一步将段内地址分成页号和偏移量，页号通过页表找到对应的物理页框，偏移量决定具体位置。

**通俗例子：** 假设您在图书馆查找某本书：

* **段**：图书馆的某个区域，比如文学类。
* **页**：找到这本书的章节，如第3章。
* **页内地址**：在这个章节中找到具体的页码，比如第5页。

通过段页式结构，系统可以灵活分配内存空间，同时更好地管理内存资源。

**第2页：IA-32架构下的内存管理**

**内容解读：** IA-32架构的内存管理包括分段和分页两个部分：

1. **分段**：CPU生成逻辑地址，并将其交给分段单元，生成线性地址。
2. **分页**：分段单元生成的线性地址再交给分页单元，将其转换为物理地址。

这种两级转换方法使得IA-32架构在内存管理上更加灵活和高效。

**通俗例子：** 类似于先定位到城市（分段），然后找到具体的街道（分页），最终确定到一个具体的地址。通过分段和分页的结合，内存地址管理更加精准和高效。

**第3页：IA-32架构的分页（两级分页）**

**内容解读：** 这页展示了IA-32的两级分页方式：

1. 使用页目录和页表两层结构，每层对应一个4KB的页面。
2. **页目录**：存储多个页表的地址，页目录寄存器 (CR3) 指向当前使用的页目录。
3. **页表**：每个页表对应4KB的页框。

这种两级结构让系统可以管理更大的地址空间。

**通俗例子：** 就像查找一个地址，先找到一本目录（页目录），再通过目录找到具体的章节（页表），最后找到特定的页。分两级结构让查找更快速，更便于管理大量的内存地址。

**第4页：IA-32架构的分页（三级分页）**

**内容解读：**

* 当面对64位硬件平台时，IA-32的两级分页已不再适用，因此引入三级分页结构。
* **三级分页**：在两级基础上添加一个“页目录指示表”，每个目录可以管理更大的地址空间。

三级分页扩展了物理地址的可访问范围，使得系统可以管理更大的物理内存。

**通俗例子：** 如果把原来的两级目录比作图书馆的图书分类，现在增加了一个总类分区（页目录指示表），可以涵盖更多的书籍类别（地址范围）。这样，通过三级目录的设计，系统可以管理海量的内存资源。

**题目 1**

**已知一个进程的逻辑地址空间分为3个段，各段的基址和长度如下：**

| **段号** | **基址** | **长度** |
| --- | --- | --- |
| 0 | 1000 | 400 |
| 1 | 2000 | 300 |
| 2 | 3000 | 500 |

**如果逻辑地址为 (1, 150)，计算对应的物理地址。**

**解答：**

1. 首先，根据段号1查找段表，获取段1的基址为2000，长度为300。
2. 检查偏移量150是否小于该段的长度300。因为150 < 300，说明偏移量有效。
3. 计算物理地址：物理地址 = 段基址 + 偏移量 = 2000 + 150 = 2150。

**答案：物理地址为2150。**

**题目 2**

**假设有一个段页式存储管理系统，内存页大小为1KB。某进程的段表如下，每个段包含的页表基址也给出：**

| **段号** | **段长度（页数）** | **页表基址** |
| --- | --- | --- |
| 0 | 4 | 1024 |
| 1 | 2 | 2048 |
| 2 | 3 | 3072 |

**若逻辑地址为 (2, 1, 512)，求对应的物理地址。**

* (2, 1, 512) 表示段号为2，页号为1，页内偏移为512。

**解答：**

1. 从段表中查找段2的信息，段2的页表基址为3072，段的长度为3页。
2. 检查页号是否小于段的长度，1 < 3，页号有效。
3. 查找段2的页表（基址为3072）中的第1页的物理页框。假设页表内容如下：
   * 页表基址3072内容：3072 -> 4000, 4004, 4008
   * 页号1对应的物理页框基址为4004。
4. 计算物理地址：物理地址 = 页框基址 + 页内偏移 = 4004 + 512 = 4516。

**答案：物理地址为4516。**

**题目 3**

**在一个段页式存储管理系统中，已知段表如下，页大小为512字节。**

| **段号** | **段长度（字节）** | **页表基址** |
| --- | --- | --- |
| 0 | 1024 | 6000 |
| 1 | 1536 | 7000 |

**假设逻辑地址为 (1, 2, 300)，求物理地址。**

* (1, 2, 300) 表示段号为1，页号为2，页内偏移为300。

**解答：**

1. 从段表中查找段1的信息，段1的页表基址为7000，段长度为1536字节。
2. 确定段1有多少页：1536 / 512 = 3 页。
3. 检查页号是否在段长度范围内，2 < 3，页号有效。
4. 假设段1的页表内容如下：
   * 页表基址7000内容：7000 -> 8000, 8500, 9000
   * 页号2对应的物理页框基址为9000。
5. 计算物理地址：物理地址 = 页框基址 + 页内偏移 = 9000 + 300 = 9300。

**答案：物理地址为9300。**

**题目 4**

**在段页式存储管理中，假设页大小为1KB，一个段的起始地址是2000。该段的页表内容如下：**

| **页号** | **页框号** |
| --- | --- |
| 0 | 5 |
| 1 | 9 |
| 2 | 4 |
| 3 | 12 |

**如果逻辑地址是 (0, 2, 500)，计算其物理地址。**

* (0, 2, 500) 表示段号为0，页号为2，页内偏移为500。

**解答：**

1. 查找段号0的起始地址2000，进入段0的页表。
2. 根据页号2，找到对应的物理页框号4。
3. 计算物理地址。页大小为1KB，所以物理页框4的起始地址是4 \* 1024 = 4096。
4. 物理地址 = 页框基址 + 页内偏移 = 4096 + 500 = 4596。

**答案：物理地址为4596。**

**题目 5**

**假设一个段页式存储系统中，页大小为4KB。某进程的段表如下，每段的页表基址和段长度（页数）如下：**

| **段号** | **页表基址** | **段长度（页数）** |
| --- | --- | --- |
| 0 | 4000 | 5 |
| 1 | 8000 | 3 |

**逻辑地址 (0, 4, 1024) 对应的物理地址是多少？**

* (0, 4, 1024) 表示段号为0，页号为4，页内偏移为1024。

**解答：**

1. 查找段0的信息，页表基址为4000，段长度为5页。
2. 确认页号4是否在段长度内，4 < 5，因此页号有效。
3. 查找段0的页表（基址为4000）中页号4的物理页框。假设页表内容如下：
   * 页表基址4000内容：4000 -> 5000, 5500, 6000, 6500, 7000
   * 页号4对应的物理页框基址为7000。
4. 计算物理地址：物理地址 = 页框基址 + 页内偏移 = 7000 + 1024 = 8024。

**答案：物理地址为8024。**

**题目 1**

**某系统采用段页式存储管理，页大小为2KB。某进程的段表如下，每个段的页表基址和段长度（页数）已给出：**

| **段号** | **页表基址** | **段长度（页数）** |
| --- | --- | --- |
| 0 | 3000 | 6 |
| 1 | 5000 | 4 |
| 2 | 8000 | 5 |

**如果逻辑地址为 (2, 3, 1500)，计算对应的物理地址。**

**解答：**

1. 查找段号2的段表信息，页表基址为8000，段长度为5页。
2. 检查页号3是否小于段的长度5，3 < 5，页号有效。
3. 假设段2的页表内容如下：
   * 页表基址8000内容：8000 -> 9000, 9500, 10000, 10500, 11000
   * 页号3对应的物理页框基址为10500。
4. 计算物理地址：物理地址 = 页框基址 + 页内偏移 = 10500 + 1500 = 12000。

**答案：物理地址为12000。**

**题目 2**

**一个段页式存储系统中，页大小为1KB。某进程的段表如下：**

| **段号** | **页表基址** | **段长度（页数）** |
| --- | --- | --- |
| 0 | 6000 | 3 |
| 1 | 6500 | 5 |
| 2 | 7000 | 4 |

**若逻辑地址为 (1, 4, 750)，求对应的物理地址。**

**解答：**

1. 查找段号1的段表信息，页表基址为6500，段长度为5页。
2. 检查页号4是否小于段的长度5，4 < 5，页号有效。
3. 假设段1的页表内容如下：
   * 页表基址6500内容：6500 -> 12000, 12500, 13000, 13500, 14000
   * 页号4对应的物理页框基址为14000。
4. 计算物理地址：物理地址 = 页框基址 + 页内偏移 = 14000 + 750 = 14750。

**答案：物理地址为14750。**

**题目 3**

**在段页式存储管理系统中，页大小为4KB。假设某进程的段表如下：**

| **段号** | **页表基址** | **段长度（页数）** |
| --- | --- | --- |
| 0 | 2000 | 4 |
| 1 | 3000 | 6 |
| 2 | 5000 | 8 |

**逻辑地址 (0, 2, 2000) 对应的物理地址是多少？**

**解答：**

1. 查找段号0的段表信息，页表基址为2000，段长度为4页。
2. 确认页号2是否小于段的长度4，2 < 4，因此页号有效。
3. 假设段0的页表内容如下：
   * 页表基址2000内容：2000 -> 4000, 4500, 5000, 5500
   * 页号2对应的物理页框基址为5000。
4. 计算物理地址：物理地址 = 页框基址 + 页内偏移 = 5000 + 2000 = 7000。

**答案：物理地址为7000。**

**题目 4**

**一个段页式存储管理系统中，页大小为2KB。假设某进程的段表如下：**

| **段号** | **页表基址** | **段长度（页数）** |
| --- | --- | --- |
| 0 | 1000 | 3 |
| 1 | 2000 | 5 |
| 2 | 4000 | 6 |

**如果逻辑地址为 (1, 3, 1800)，计算对应的物理地址。**

**解答：**

1. 查找段号1的段表信息，页表基址为2000，段长度为5页。
2. 确认页号3是否小于段的长度5，3 < 5，因此页号有效。
3. 假设段1的页表内容如下：
   * 页表基址2000内容：2000 -> 8000, 8500, 9000, 9500, 10000
   * 页号3对应的物理页框基址为9500。
4. 计算物理地址：物理地址 = 页框基址 + 页内偏移 = 9500 + 1800 = 11300。

**答案：物理地址为11300。**

**题目 5**

**一个段页式存储管理系统中，页大小为1KB。已知段表和对应页表内容如下：**

| **段号** | **页表基址** | **段长度（页数）** |
| --- | --- | --- |
| 0 | 2500 | 4 |
| 1 | 3000 | 3 |
| 2 | 4000 | 2 |

页表内容如下：

* 页表基址2500内容：2500 -> 1000, 2000, 3000, 4000
* 页表基址3000内容：3000 -> 5000, 6000, 7000
* 页表基址4000内容：4000 -> 8000, 9000

**求逻辑地址 (0, 2, 500) 的物理地址。**

**解答：**

1. 查找段号0的段表信息，页表基址为2500，段长度为4页。
2. 检查页号2是否小于段长度4，2 < 4，因此页号有效。
3. 根据段号0的页表内容，页号2对应的物理页框基址为3000。
4. 计算物理地址：物理地址 = 页框基址 + 页内偏移 = 3000 + 500 = 3500。

**答案：物理地址为3500。**